Thambipillai Srikanthan
Jingling Xue
Chip-Hong Chang (Eds.)

Advances in Computer Systems Architecture

LNCS 3740

10th Asia-Pacific Conference, ACSAC 2005
Singapore, October 2005
Proceedings

Springer

Lecture Notes in Computer Science 3740

Commenced Publication in 1973
Founding and Former Series Editors:
Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

Thambipillai Srikanthan Jingling Xue
Chip-Hong Chang (Eds.)

Advances in Computer Systems Architecture

10th Asia-Pacific Conference, ACSAC 2005
Singapore, October 24-26, 2005
Proceedings

Springer

Volume Editors

Thambipillai Srikanthan
Nanyang Technological University, School of Computer Engineering
Blk N4, Nanyang Avenue, Singapore, 639798
E-mail: astsrikan@ntu.edu.sg

Jingling Xue
University of New South Wales, School of Computer Science and Engineering
Sydney, NSW 2052, Australia
E-mail: jxue@cse.unsw.edu.au

Chip-Hong Chang
Nanyang Technological University, School of Electrical and Electronic Engineering
Blk S2, Nanyang Avenue, Singapore 639798
E-mail: echchang@ntu.edu.sg

Library of Congress Control Number: 2005934301

CR Subject Classification (1998): B.2, B.4, B.5, C.2, C.1, D.4

ISSN 0302-9743
ISBN-10 3-540-29643-3 Springer Berlin Heidelberg New York
ISBN-13 978-3-540-29643-0 Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 11572961 06/3142 5 4 3 2 1 0

Preface

On behalf of the Program Committee, we are pleased to present the proceedings of the 2005 Asia-Pacific Computer Systems Architecture Conference (ACSAC 2005) held in the beautiful and dynamic country of Singapore. This conference was the tenth in its series, one of the leading forums for sharing the emerging research findings in this field.

In consultation with the ACSAC Steering Committee, we selected a 33-member Program Committee. This Program Committee represented a broad spectrum of research expertise to ensure a good balance of research areas, institutions and experience while maintaining the high quality of this conference series. This year's committee was of the same size as last year but had 19 new faces.

We received a total of 173 submissions which is 14% more than last year. Each paper was assigned to at least three and in some cases four Program Committee members for review. Wherever necessary, the committee members called upon the expertise of their colleagues to ensure the highest possible quality in the reviewing process. As a result, we received 415 reviews from the Program Committee members and their 105 co-reviewers whose names are acknowledged in the proceedings. The conference committee adopted a systematic blind review process to provide a fair assessment of all submissions. In the end, we accepted 65 papers on a broad range of topics giving an acceptance rate of 37.5%. We are grateful to all the Program Committee members and the co-reviewers for their efforts in completing the reviews within a tight schedule.

In addition to the contributed papers, this year's program included two keynote speeches from authorities in academia and industry: Ruby B. Lee of Princeton University on *Processor Architecture for Trustworthy Computers*, and Jesse Z. Fang of Intel Corporation on *Challenges and Opportunities on Multi-core Microprocessor*.

It was a rewarding experience to be the Program Chairs for this year's conference. We wish to take this opportunity to thank many people who contributed to making ACSAC 2005 a great success. Firstly, we thank the authors for submitting their work to this year's conference. We thank our efficient and energetic Organizing Committee. In particular, we would like to thank the Publicity Chairs, Vinod Prasad and Tulika Mitra, for having done a wonderful job in publicizing this conference and attracting a high number of submissions, the Web Chairs, Jiajia Chen and Xiaoyong Chen, for maintaining the online conference Web pages, and the Local Arrangements Chair, Douglas Maskell, for ensuring the smooth running of the conference in Singapore. We thank all the Program Committee members, who contributed considerable amounts of their valuable time. It was a great pleasure working with these esteemed members of our research community. We also thank all our sponsors for their support of this event.

Last, but not least, we would like to thank the General Chair, Graham Leedham, for his commitment and perseverance in this invaluable role.

We sincerely hope you will find these proceedings valuable and look forward to your participation in future ACSAC conferences.

August 2005

Thambipillai Srikanthan
Jingling Xue
Chip-Hong Chang

Conference Organization

General Chair

Graham Leedham	Nanyang Technological University, Singapore

Program Chairs

Thambipillai Srikanthan	Nanyang Technological University, Singapore
Jingling Xue	University of New South Wales, Australia

Publications Chair

Chip-Hong Chang	Nanyang Technological University, Singapore

Publicity Chairs

Vinod Prasad	Nanyang Technological University, Singapore
Tulika Mitra	National University of Singapore, Singapore

Local Arrangements Chair

Douglas Maskell	Nanyang Technological University, Singapore

Web Chairs

Jiajia Chen	Nanyang Technological University, Singapore
Xiaoyong Chen	Nanyang Technological University, Singapore

Program Committee

K. Vijayan Asari	Old Dominion University, USA
Eduard Ayguade	UPC, Spain
Sangyeun Paul Cho	University of Pittsburgh, USA
Lynn Choi	Korea University, Korea
Christopher T. Clarke	University of Bath, UK
Oliver Diessel	University of New South Wales, Australia
Jean-Luc Gaudiot	University of California, Irvine, USA
James Goodman	University of Auckland, New Zealand
Gernot Heiser	National ICT, Australia
Hock Beng Lim	National University of Singapore, Singapore
Wei-Chung Hsu	University of Minnesota, USA
Chris Jesshope	Universiteit van Amsterdam, Netherlands
Hong Jiang	University of Nebraska, Lincoln, USA
Sridharan K.	Indian Institute of Technology, Madras, India
Feipei Lai	National Taiwan University, Taiwan
Xiang Liu	Peking University, China
Balakrishnan M.	Indian Institute of Technology, Delhi, India
Philip Machanick	University of Queensland, Australia
John Morris	University of Auckland, New Zealand
Tadao Nakamura	Tohoku University, Japan
Sukumar Nandi	Indian Institute of Technology, Guwahati, India
Tin-Fook Ngai	Intel China Research Center, China
Andrew P. Paplinski	Monash University, Australia
Lalit M. Patnaik	Indian Institute of Science, India
Jih-Kwon Peir	University of Florida, USA
Damu Radhakrishnan	State University of New York, USA
Rajeev Thakur	Argonne National Laboratory, USA
Tanya Vladimirova	University of Surrey, Guildford, UK
Weng-Fai Wong	National University of Singapore, Singapore
Chengyong Wu	Institute of Computing Technology, CAS, China
Yuanyuan Yang	State University of New York at Stony Brook, USA
Pen-Chung Yew	University of Minnesota, USA
Weimin Zheng	Tsinghua University, China

Co-reviewers

Pete Beckman
Dan Bonachea
Dmitry Brodsky
Darius Buntinas
Bin Cao
Francisco Cazorla
Ernie Chan
Yen Jen Chang
Howard Chen
William Chen
Kuen-Cheng Chiang
Archana Chidanandan
Young-ll Cho
Peter Chubb
Josep M. Codina
Leena D.
Xiaoru Dai
Abhinav Das
Ryusuke Egawa
Kevin Elphinstone
Rao Fu
Zhiguo Ge
Gabriel Ghinita
Qian-Ping Gu
Hui Guo
Yajuan He
Sangjin Hong
Shen Fu Hsiao
Sun-Yuan Hsien
Wei Hsu
Lei Huang
Wei Huo
Andhi Janapsatya
Yaocang Jia
Gui Jian
Priya T.K.
Mahmut Kandemir
Jinpyo Kim
Lin Wen Koh
Shannon Koh
Anoop Kumar Krishna
Mong-Kai Ku
Hiroto Kukuchi
Edmund Lai Ming-Kit
Robert Latham
Jonghyun Lee
Sanghoon Lee
Jussipekka Leiwo
Simon Leung
Xiaobin Li
Xueming Li
Jen-Chiun Lin
Jin Lin
Chen Liu
Lin Liu
Shaoshan Liu
Xuli Liu
Jiwei Lu
Yujun Lu
Ming Ma
Usama Malik
Verdi March
Xavier Martorell
Guillaume Mercier
Nader Mohamed
Enric Morancho
Arun Nair
Mrinal Nath
Hau T. Ngo
Deng Pan
Kaustubh S. Patkar
Kolin Paul
Jorgen Peddersen
Marius Portmann
Daniel Potts
Felix Rauch
Tom Robertazzi
Shang-Jang Ruan
Sergio Ruocco
Esther Salami
Chunlei Sang
Olivero J. Santana
Seng Lin Shee
Menon Shibu
Mon-Chau Shie
David Snowdon

Dan Sorin
Ken-ichi Suzuki
Brian Toonen
Patchrawat Uthaisombut
Venka
Kugan Vivekanandarajah
Shengyue Wang
Yiran Wang
Hui Wu
Bin Xiao
Chia-Lin Yang
Hongbo Yang
Min Yang
Kiren Yellajyosu
Kyueun Yi
Antonia Zhai
Ming Z. Zhang
Yifeng Zhu

Table of Contents

Session 2A: Processor Architectures and Microarchitectures

Session 2B: High-Reliability and Fault-Tolerant Architectures

Session 3A: Compiler and OS for Emerging Architectures

Session 3B: Data Value Predictions

Keynote Address II

Session 4A: Reconfigurable Computing Systems and Polymorphic Architectures

Session 4B: Interconnect Networks and Network Interfaces

Session 5A: Parallel Architectures and Computation Models

Session 5B: Hardware-Software Partitioning, Verification, and Testing of Complex Architectures

Session 6A: Architectures for Secured Computing

Session 6B: Simulation and Performance Evaluation

Session 7: Architectures for Emerging Technologies and Applications I

Session 8: Memory Systems Hierarchy and Management

Session 9: Architectures for Emerging Technologies and Applications II

Processor Architecture for Trustworthy Computers

Ruby B. Lee

Forrest G. Hamrick Professor of Engineering and
Professor of Electrical Engineering,
Princeton University
rblee@princeton.edu

We propose that computer architects need to design more trustworthy computers that protect a user's information, computations and communications from attacks by malicious adversaries. This is in addition to providing current engineering goals of higher performance, lower cost, lower power consumption and smaller footprint.

Today, a user can trust his computer to do what he asks, correctly and efficiently. However, he has very little assurance that the computer will *not* do anything else, and that the computer will not be controllable by an adversary intent on performing malicious acts. The proliferation of interconnected computing devices using publicly accessible Internet and wireless networks increases these threats, since attackers need not even have physical access to the computing device. The programmability of such ubiquitous computing devices further increases these threats, since complex software often have many security vulnerabilities that may be exploited by an attacker.

A trustworthy computer is one that has been designed with features built in to protect the user from malicious attackers and other threats. For example, a user would like to be able to trust his computer to protect him from attackers who may want to expose, modify or destroy sensitive information he has stored either in his computer or on on-line storage accessible from his computer [1]. He would like protection from attackers who want to use his computer to attack others [2, 3], to spy on his computations and communications, to inject viruses, worms and other malware into his machine, or to prevent him from using his own computer or resources accessible to him through his computing device. In future trustworthy computers, a user should be given some assurance that these security concerns were addressed in the design of his computing device.

The design of a trustworthy computer requires a very different design approach, which we call "threat-based design". This is in contrast to, and in addition to, conventional functionality-based or performance-based design. Threat-based design requires, first, an understanding of the threats being addressed. Then, it requires the inclusion of mechanisms to prevent, detect, mitigate or stop the attack, or somehow render it harmless. Computer architects can approach threat-based design in two main directions: a clean-slate design of a new secure and trustworthy architecture for the processor, platform and system, or a compatibility-based design that adds security features to existing processors, platforms and systems.

T. Srikanthan et al. (Eds.): ACSAC 2005, LNCS 3740, pp. 1–2, 2005.

In this talk, we will give a few examples of threats that a trustworthy computer should defend against, and how this threat-based design can translate into concrete processor architectural features. We also discuss principles and insights gleaned from these examples.

For example, we will discuss a new trust model which creates a small island of trust, with most of the hardware and software being untrusted. We show how this island of trust can be supported with a small set of new processor architecture features that can be added to any existing processor or integrated into a clean-slate design of a new trustworthy processor. These features in a microprocessor provide "virtual secure co-processing", i.e., the microprocessor itself acts as a secure coprocessor when needed. We also call this "Secret Protected (SP)" architecture, since the island of trust created can be used to protect a user's critical secrets from exposure, corruption and other attacks [1]. We show an application where a user can securely and conveniently access, store and transmit sensitive information on public networks using different SP-enabled computing devices. The sensitive information, whether program, data or files, is encrypted and SP architecture provides protection for the user's key chain, with hardware support in the processor for protecting the master keys. This concrete example demonstrates that security can be provided without compromising performance, cost or ease-of-use.

We strongly advocate that secure and trustworthy computers be designed in conjunction with traditional design goals like performance, cost, power consumption and usability, since otherwise they will not be widely deployed. Certainly tradeoffs have to be made, but the challenge is to provide secure and trustworthy operation in a high performance computer that is convenient and easy to use. We encourage computer architects and hardware and software designers to apply their creative talents to this formidable, important and exciting challenge. Indeed, it is high time that the computer architecture community joins the "arms race" by proposing innovative yet pragmatic architectures that can thwart the epidemic escalation of security breaches in cyberspace.

References

[1] Ruby B. Lee, Peter C. S. Kwan, John P. McGregor, Jeffrey Dwoskin, and Zhenghong Wang, "Architecture for Protecting Critical Secrets in Microprocessors", *Proceedings of the 32nd International Symposium on Computer Architecture*, pp. 1-12, June 2005.

[2] Ruby B. Lee, David K. Karig, John P. McGregor, and Zhijie Shi, "Enlisting Hardware Architecture to Thwart Malicious Code Injection", *Proceedings of the International Conference on Security in Pervasive Computing (SPC-2003)*, LNCS 2802, pp. 237-252, Springer Verlag, March 2003.

[3] Stephen M. Specht and Ruby B. Lee, "Distributed Denial of Service: Taxonomies of Attacks, Tools, and Countermeasures", *Proceedings of the 17th International Conference on Parallel and Distributed Computing Systems, International Workshop on Security in Parallel and Distributed Systems*, pp. 543-550, September 2004.

Efficient Voltage Scheduling and Energy-Aware Co-synthesis for Real-Time Embedded Systems

Amjad Mohsen and Richard Hofmann

Department of Communication Systems and Computer Networks,
University of Erlangen, Martensstr. 3, 91058 Erlangen, Germany
Amjad.muhsen@infromatik.uni-erlangen.de, rhofmann@cs.fau.de

Abstract. This paper presents an integrated methodology and a tool for system-level low power/energy co-synthesis for real-time embedded systems. Voltage scheduling (VS) is being applied to utilize the inherent slacks in the system. The voltage schedule is generated based on a global view of all tasks' mapping and their energy profiles. The tool explores the three dimensional design space (performance-power-cost) to find implementations that offer the best trade-off among these design objectives. Unnecessary power dissipation is prevented by refining the allocation/binding in an additional synthesis step. The experimental results show that our approach remarkably improves the efficiency of VS and leads to additional energy savings, especially for applications with stringent delay constraints.

1 Introduction

Power consumption is one of the major challenges that face nearly all types of present and future battery-operated and digital embedded systems. Reducing the power/energy dissipation makes these systems more competitive and leads to longer battery lifetime. At the same time, packaging and cooling expenses are directly related to the power/energy dissipation in all digital systems. Therefore, without an integrated methodology that can sharply reduce power consumption, mobile electronics will suffer from short operation periods or heavy battery weights.

Reducing the power/energy consumption in digital systems has been an area of active research. Related issues have been investigated and handled at different abstraction levels. Low level methodologies supported by CAD tools, such as SPICE can barely save half of the consumed power. This is related to the fact that decisions made at these levels have only limited and local effect on the consumed power. Moreover, modifying the design at these levels causes longer design cycle since different design decisions have already been taken at higher levels of abstraction. Therefore, high level tools are motivated which can reduce the design cycle significantly. These tools can also lead to design alternatives that are more efficient from the power/energy point of view.

Typically, tackling power issues at the highest possible abstraction level has the most global effect. So, using integrated and automated co-design methodologies starting at the system level is recommended to achieve drastic power reduction and better system optimization. It is worth noticing that system level methodologies are applied as a preliminary step for low power system optimization that can be combined with other low level approaches.

T. Srikanthan et al. (Eds.): ACSAC 2005, LNCS 3740, pp. 3 – 14, 2005.

However, automated co-design setup at high abstraction levels needs at least two supporting requirements: Firstly, a specification language that supports automated implementation and verification of functional and temporal behaviour of real-time embedded systems [1]. Secondly, the required information for performing such automated implementation has to be abstracted from low levels and supplied at the intended high level where it can be used by automated synthesis and optimization tools [2].

An efficient scheme that can reduce the consumed power and achieve energy-efficient computing is dynamic voltage scaling (DVS). Here, the supply voltage can be reduced on demand to satisfy the required rather than the desired performance. This in return causes a quadratic energy reduction without sacrificing peak performance of the system. At high levels of abstraction, the voltage level(s) required to execute each task can be statically planned for applications that have predictable loads and predetermined limits on computation performance. Considering power profiles of the allocated components when scaling the voltage is a source of extra power/energy reduction. This is related to the fact that the higher the energy consumption of a task the more energy saving it causes when scaling the supply voltage.

The remainder of this paper is organized as follows: Section 2 presents a summary of selected related work in the area of power/energy minimization and design space exploration. Section 3 presents our automated co-synthesis methodology for low power/energy. The experimental results are presented in section 4. We conclude in section 5 and suggest some issues to be handled in future work.

2 Related Work

In recent years, tools have been devised for exploring the design space at high abstraction levels. Thiele et al. have suggested a system level design space exploration methodology for architectures of packet processing devices [3]. An evolutionary-based approach for system level synthesis and design space exploration was suggested in [4]. Slomka et al. have presented a tool for hardware/software co-design of complex embedded systems with real-time constraints, CORSAIR [5]. The co-synthesis process was based in CORSAIR on a three-level tabu search algorithm. The above mentioned approaches did not handle the power problem at all or did not tackle it concretely.

A power estimation framework for hardware/software System-on-Chip (SoC) designs was introduced in [6]. The approach was based on concurrent execution of different simulators for different parts of the system (hardware and software parts). Although this approach could fairly be accurate it is very slow, especially for large systems when a huge number of design alternatives is available.

Hybrid search strategies (global/local) for power optimization in embedded DVS-enabled multiprocessors were introduced in [7]. This approach used local optimization based on hill climbing and Monte Carlo search inside a genetic-based global optimization. Although this approach could yield the required voltage levels that minimize the energy per computation period, it could be very time consuming, especially when applied at high abstraction levels. In addition, the influence of power profiles of the tasks was not included when deriving the voltage schedule.

Gruian has introduced two system level low-energy design approaches based on DVS-enabled processors [8]. The first was based on performance-energy tradeoffs whereas the second was based on energy sensitive scheduling and mapping techniques. In this approach, simulated annealing was used for generating task-processor mappings.

An energy conscious scheduling method was introduced in [9]. This methodology assumed a given allocation and tasks-processors assignment (DVS-enabled processors). The energy was minimized by selecting the best combination of supply voltage levels for each task executing on its processor.

A low power co-synthesis tool (LOPOCOS) was suggested in [10] and assumed DVS-enabled architectures. The objective was to help the designer to identify an energy-efficient application partitioning for embedded systems implemented using heterogeneous distributed architectures. Although it performs better than previously suggested approaches, for applications with stringent delay constraints, LOPOCOS has moderate reduction influence on the consumed power/energy. Additionally, component allocation in LOPOCOS was user driven and was therefore based on designers' knowledge and experience. It was not integrated in the optimization loop to automate the whole design process.

Another energy-efficient co-design approach was presented in [11]. This approach is similar to the above mentioned one, but the allocation was automatically optimized in this approach during the co-synthesis process. An additional allocation/refinement step was proposed to further optimize the design. For that purpose, power- and performance-optimized components' types were suggested. The influence of using power optimized components on the overall power consumption was presented. The maximum achieved power reduction was about 20% for the included benchmarks in the study.

Many of the previously introduced approaches dealt with the power problem at high abstraction levels and utilized the power-performance tradeoffs by using DVS-enabled architectures. However, the following issues were not yet solved satisfactorily: 1) The special needs of optimizing the co-synthesis process when applying DVS. 2) For applications with stringent performance constraints, DVS may even fail to cause significant power/energy reductions. 3) The combined effect of using different components' types and voltage scaling was not addressed at all.

Our proposed methodology for low power/energy co-design deals with the issues mentioned above. Starting at the level of FDTs (formal description techniques), the tool is able to explore the available design space while handling design tradeoffs. It yields low cost and power optimized implementation(s) under pre-defined stringent performance limits. The voltage schedule is static and based on a global view of energy profiles of the tasks and their mappings. The integrated library is enhanced by a set of special features to enable fast design space exploration and to improve the efficiency of VS. Combining these issues together in one system-level tool leads to drastic power/energy reduction, especially for real-time systems with stringent design constraints. This paper tackles at the first place the dynamic power consumed in embedded systems but the proposed algorithms are general and can be extended to handle issues related to static power.

3 Design Flow and Design Space Exploration

To be able to handle the complexity of designing large embedded systems with the presence of hard time constraints, the design process is decomposed in our co-design methodology into four phases: System specification, co-synthesis, implementation synthesis, and evaluation and validation. These steps are briefly described below before explaining our voltage scheduling methodology.

3.1 Design Phases

The overall automated co-design methodology consists of the following steps:

3.1.1 System Specification

This phase transforms the informal specifications into formal specifications. We use the SDL/MSC which is one of the prominent and successfully applied techniques in telecommunication industry [12]. SDL (specification and description language) is used to describe the functional specification. MSC (message sequence chart) is extended to describe timing requirements and other non-functional aspects.

3.1.2 Co-synthesis

An internal system model which includes a problem graph (PG) and an architecture graph (AG) are automatically generated from the specification. The PG is a directed acyclic graph $Fp(\Psi,\Omega)$, where Ψ represents the set of vertices in the graph ($\psi_i \in \Psi$) and Ω is the set of directed edges representing the precedence constraints ($\omega_i \in \Omega$). The AG is $FA(\Theta,\Re)$, where Θ represents the available architectures ($\theta_i \in \Theta$) and ($\rho_i \in \Re$) represents the available connections between hardware components. For each hardware component ($\theta_i \in \Theta$), a finite set of resource types (S) is defined. For each resource type ($s_i \in S$) there is a set of associated ratios (R_s) that specify power, delay, and cost scaling when using this type for a selected component.

The automated co-synthesis methodology optimizes the allocation, binding and scheduling (time and voltage). The cost of the final implementation and its performance as well as the amount of consumed power/energy are considered during optimization. So, the co-synthesis can be seen as a multi-objective optimization problem that searches the design space to find implementations that satisfy design constraints. The search-space engine we present in this article is based on an up-to-date evolutionary algorithm (see section 3.3). Evolutionary algorithms are able to process a set of different implementation candidates at the same time. This inherent parallelism made evolutionary algorithms suitable for optimization problems which have complex and large search spaces. In Fig. 1 the basic steps in the global optimization algorithm are presented. Power estimation and evaluation in our approach are based on a library of pre-characterized components.

The components' library offers hardware and software components of different types. Also, components of different granularities are modelled. These features improve the performance of exploring the design space as well as the estimation accuracy. Estimating the power consumed by a design alternative is performed by combining the number of accesses to each allocated component with the power model of that component. The power model of each component is loaded from the library.

Input: $F_p(\Psi,\Omega)$, $F_A(\Theta,\Re)$, *technology library*
Output: *allocation/binding, schedule (time and voltage)*

Step 0: Generate initial population.
Step 1: Decode implementation.
Step 2: Repair infeasible implementations.
Step 3: Evaluate and refine each implementation:
- Compute a time schedule (*if any*).
- Refine the allocation/binding.
- Compute a *voltage schedule (Figure 2).*
- Compute objective values.
- Force penalty to reflect design constraints violation.

Step 4: Check termination (design constraints).
Step 5: Assign fitness and perform selection (SPEA2).
- Environmental selection (archive update)
- Mating selection (produce the mating pool)

Step 6: Variation: recombination operators
(Crossover & mutation)
Go to Step 1.

Fig. 1. Global optimization algorithm

Each individual in the population represents a candidate implementation. The allocation/binding refinement step refines the allocation and binding to handle power-performance-cost tradeoffs in a better way. This step deals with an ordered list of types for each component and leads to allocating performance optimized instances to execute critical-path tasks and power optimized ones for non-critical path tasks. A new schedule is generated after the refinement step.

Since we assume DVS-enabled architectures, the scheduling issue in this case is transformed into a two dimensional problem: Time and voltage. A list-based scheduler performs the time scheduling, whereas the voltage schedule is computed in such a way that the available slack is utilized efficiently without violating performance constraints. The computed voltage schedule is stored in a table-like form, which keeps the overhead of voltage scheduling during run-time at minimum.

3.1.3 Implementation Synthesis and Evaluation and Validation

Commercial tools and our own SDL compiler are used for translating the SDL specifications into software implementation in C and hardware implementations in VHDL. Compilation for VHDL and C is carried out by commercial tools, which are readily available from many vendors.

3.2 Applying VS

For applications that have predictable computational loads with a pre-determined upper constraint on performance, it is possible to estimate the benefits of VS [13]. However, applying VS introduces two new overheads: Transition time and transition

energy which represent the required time and energy for changing the voltage from *level1* to *level2*, respectively [14]. The overhead of applying VS is considered in our methodology and assumptions related to this overhead (energy and cost) are taken from [7].

Reducing the supply voltage has another serious influence on circuit performance. Scaling down the supply voltage increases the circuit delay. Therefore, the voltage may only be reduced if the corresponding degradation in performance can be tolerated. Assuming that performance degradation is acceptable when executing a given task (ψ_i) (*based on the given delay constrains*), the energy consumed by this task ($E'(\psi_i)$) at a voltage level V_{level}, can be calculated as follows:

$$E'(\psi_i) = \left(\frac{V_{level}^2}{V_{supply}^2} \right) E(\psi_i) \Big|_{V=V_{supply}} \quad (1)$$

In the above equation, V_{supply} is the nominal supply voltage ($V_{level} \leq V_{supply}$) and $E(\psi_i)$ refers to the energy consumed by the corresponding task at the nominal voltage.

Input: $F_p(\Psi,\Omega)$, $F_A(\Theta,\Re)$, *mapping, time schedule, step*.
Output: *Voltage schedule* $V_{ss}(t)$

Step 1:
- Calculate ΔEN_i of all tasks $\psi_i \in \Psi$
- Assign $P_{priority}$ to all tasks $\psi_i \in \Psi$
- Create empty list *LS* of size *y*

Step 2:
Arrange the tasks in *LS* in a descending order of $P_{priority}$.

Step 3:
Get ψ_j that has the highest non-zero $P_{priority}$ from *LS*.
- If (V_{dd} is no longer > $2V_t$) → remove ψ_j from *LS*.
- Else, extend the task (ψ_j) in steps of (*n*step*).
- Update the tasks profile and propagate delay effects.

Step 4:
Return if *LS* is empty OR all tasks have $P_{priority} = 0$

Step 5:
- Calculate ΔEN of all tasks in LS.
- Assign $P_{priority}$ to all tasks.
- Go to step 2.

Fig. 2. Voltage scheduling algorithm

The time needed to execute a task is increased when this task is executed at a voltage level lower than the maximum. This in turn may affect (increase or decrease) the available slack or idle time interval for other tasks which are not necessarily

mapped to the same hardware. This is related to the mapping of these tasks and the precedence constraints between tasks. So, in order to take into consideration this inter-task relation, we perform the voltage level planning based on a global view of all tasks and their energy profiles. The voltage scheduling algorithm is depicted in Fig. 2. In this figure, (y) refers to the number of tasks and V_t is the threshold voltage. ΔEN_i refers to the energy saving for task ψ_i when extending its execution time (by one time step (Step, n = 1)) by scaling its operating voltage:

$$\Delta EN_i = E(\psi_i) - E'(\psi_i)\big|_{n=1} \tag{2}$$

The achieved energy reduction is closely related to ΔEN_i [15]. So, tasks with larger energy profile are given more preference to extend their execution. The power priority ($P_{priority}$) for task ψ_i is proportional to the calculated ΔEN_i multiplied by sl_i which is defined as:

$$sl_i = \begin{cases} 1, & slack_i \neq 0 \\ 0, & otherwise \end{cases} \tag{3}$$

The task which has the maximum effect on the energy consumption is selected firstly to extend its execution by scaling the voltage. After extending the execution of a task by reducing the supply voltage, the power value is updated for the selected task and the effect of the time extension for this task is propagated through other related tasks. The algorithm above terminates when one of two conditions is satisfied: 1) When the list LS is empty. This case occurs if the voltage level is reduced to a value around $2V_t$ for all tasks. Actually, the minimum possible value for V_{dd} is set relative to the worst case threshold voltage. A practical limit for it is about $2V_t$ [20]. 2) When $P_{priority} = 0$ for all tasks which means there is no available slack to be exploited by any of the tasks.

3.3 Evolutionary Algorithm Design

Optimization during the co-synthesis process is based in our approach on the theory of evolutionary algorithms. The widely used evolutionary multi-objective optimizer SPEA2 [16] is integrated as a ready-to-use black-box to our automated co-design framework. The optimization goal is to find design alternatives with Pareto-optimal objective vectors.

The initial population is generated randomly (see Fig. 1). Repair heuristics and penalty functions are used to deal with infeasibility problems in the generated implementations. The repair mechanism is based on a priority list of hardware components that can be allocated to execute a task.

Violating design constraints (delay, cost, and power) is handled using appropriate penalty functions. Each penalty function takes into consideration the number of violations of this design constraint's type and the distance from the feasible region. For example, the penalty function for violating power constraints in a given path is given by:

$$p(g_i,J) = \mu \cdot m \cdot \sum_{i=1}^{m} \delta_i \left(\frac{\left(P_i(\alpha,\beta) - P_{MAX_i} \right)}{P_{MAX_i}} \right) \quad (4)$$

where P_{MAX_i} is the forced power constraint on path i in the task graph, $P_i(\alpha,\beta)$ is the actual consumed power for a given allocation α and binding β, δ_i indicates how crucial violating this constraint is, m is related to the number of violations of this type of constraint, and μ is a controlling parameter. The fitness function *f(J),* when a certain design constraint is violated, is the sum of the penalty function and the objective function *h(J)*. The obtained fitness function represents the new objective function $h'(J)$. So, the problem is transformed into another problem with $f(J) = h'(J)$. The optimization process is guided by the objective function $h'(J)$ and the fitness of an implementation is calculated based on the Pareto-optimization scheme (SPEA2).

The variation process includes applying reproduction rules (crossover and mutation) to change the genetic material of individuals. Different types of crossover (such as uniform and single point) and mutation (such as one-point and independent) are implemented. The types of these operators are specified by the user. Different controlling parameters have been experimentally specified. The selection process is performed by SPEA2. The optimization loop is terminated when design constraints are satisfied or when a pre-specified number of generations is exceeded.

4 Benchmarks and Experimental Results

A set of benchmarks was used to investigate the effectiveness of our integrated methodology, part of these benchmarks are taken from real-life examples. The benchmarks themselves can be categorized into two groups: The first group includes: the one dimensional FFT, the Gaussian Elimination (GE), both taken from [17], and the Optical Flow Detection (OFD) which is part of an autonomous model of a helicopter [18]. The OFD algorithm runs originally on two DSPs with an average current of 760 mA at 3.3V. It uses a repetition rate of 12.5 frames of 78× 120 pixels per second. The original data for power consumption and performance are partially taken from [10]. The second group includes a set of benchmarks originally generated using the "Task Graphs For Free" TGFF [19].

The results are reported using the achieved energy reduction in percent. These results are categorized in three groups: The benefit of using power optimized types and architectures by using allocation/binding refinement [11], the benefit of applying VS, and the achieved benefit by combining both VS and the allocation refinement step. To clearly demonstrate the effect of allocation refinement using different hardware types on the performance of VS, additional experiments are carried out by imposing more stringent performance constraints while applying our design methodology.

4.1 Using Power Optimized Types

Column two in Table 1 shows the obtained power reduction when applying the methodology presented in [11]. The results indicate that the maximum energy reduction that can be achieved is barely exceeding 20%. Nevertheless, this additional refinement step is valuable when combined with voltage scaling. The influence of this combination is presented below in more details.

4.2 The Effect of Applying VS

The benefit of applying VS is shown in column 3 of Table 1. The minimum benefit achieved when applying VS is obtained by the GE. This can be related to the stringent performance constraints forced on this application and the data dependency in the task graph. For the first group of benchmarks, the FFT achieves an energy reduction of about 42% when applying VS whereas TGFF6 seems to make the best benefit of VS (83.5%). It is clearly seen that VS is superior to only refining allocation refinement.

4.3 Combining Allocation Refinement and VS

When delay constraints provide enough slack intervals to be utilized by the VS, the effect of refining the allocation/binding is extremely limited. TGFF1 is an example on this case. Column 4 of Table 1 shows the effect of the allocation refinement step on VS. It is clearly seen that the performance of VS is improved, but in varying degrees.

Table 1. Energy reduction in % when applying allocation refinement and VS

Benchmark	Allocation Refinement [11]	VS	VS & Allocation Refinement
TGFF1	19.7	68.1	69.1
TGFF2	14.8	36.4	45.1
TGFF3	12.7	64.6	77.1
TGFF4	10.8	82.6	88.1
TGFF5	13.1	60..1	64.9
TGFF6	18.1	83.5	93.6
TGFF7	13.8	30.2	70.4
TGFF8	20.3	76.6	78.9
TGFF9	16.7	37.3	83.6
TGFF10	2.7	19.6	49.5
FFT	17	42	66
GE	15	18	26
OFD	6	22	27

The effect of this refinement step increases and becomes clearer as the performance constraints get more and more stringent. Table 2 shows the achieved energy reduction when more stringent performance constraints are forced by scaling the original constraints. Benchmarks with stringent performance constraints are distinguished from the original ones in these experiments by a "*", (i.e., TGFF*). In order to enable fair comparison, the energy reduction obtained under these tight performance constraints when applying VS are evaluated and presented in column 2 of the same table. The last column of this table shows the joint influence of VS and the extra refinement step on the achieved energy reduction.

The results presented in Table 2 obviously show that the potential benefit of applying VS is sharply limited under tight performance constraints. The maximum energy reduction that could be achieved in this experiment is about 25% only.

It can be seen that for applications with stringent performance constraints, the effect of applying VS can noticeably be improved when combined with the proposed allocation refinement step. For example, the effect of using VS is almost tripled when combined with the additional allocation refinement step for TGFF4*, whereas the energy reduced is improved by a factor of 15 for TGFF10*. This justifies the need for this additional optimization step.

As we pointed out previously, system level power/energy optimization is not suggested to replace but to aid low level system optimization methodologies. Throughout the design process, the system will undergo other optimization steps at low abstraction levels.

Table 2. Energy reduction in % when applying allocation refinement and VS under stringent performance constraints

Benchmark	VS	VS & Allocation Refinement
TGFF1*	25.4	36.8
TGFF2*	20.4	43.6
TGFF3*	11.9	53.6
TGFF4*	22.3	61.4
TGFF5*	16.2	53.6
TGFF6*	20.0	84.9
TGFF7*	5.7	34.5
TGFF8*	17.3	76.5
TGFF9*	7.2	64.3
TGFF10*	3.1	45.3

5 Conclusions and Further Work

This paper adds a new dimension to the multi-objective optimization problem of system co-synthesis. The voltage schedule is optimized as an integrated part during

the co-synthesis process. The proposed automated and energy-efficient co-design tool for real-time embedded systems can be of valuable help for system level designers. The tool is able to explore the performance-power-cost design space. It guides the design process to low power/energy design alternatives that satisfy other design constraints.

Voltage scaling has been successfully integrated in the co-synthesis process and its performance is remarkably enhanced by using the suggested additional refinement step. All benchmarks included in this study showed the effectiveness of the presented approach.

Currently, we are developing methodologies to reduce the overhead related to switching the voltage level when applying VS. Furthermore, it is part of the future work to encode the voltage level in the gene that represents the implementation.

Acknowledgement

We are grateful indeed that DAAD (German Academic Exchange Service) supports this research since 2002.

References

1. Münzenberger, R., Dörfel, M., Hofmann, R., and Slomka, F.: A General Time Model for the Specification and Design of Embedded Real-Time Systems. *Microelectronics Journal*, vol. 34, (2003). 989-1000.
2. Mohsen, A., and Hofmann, R.: Characterizing Power Consumption and Delay of Functional/Library Components for Hardware/Software Co-design of Embedded Systems. In *the 15th IEEE International Workshop on Rapid System Prototyping (RSP'04)*, Geneva, (2004) 45-52.
3. Thiele, L., Chakraborty, S., Gries, M., and Künzli, S.: Design Space Exploration of Network Processor Architectures. In *Network Processor Design: Issues and Practices*, Vol. 1, October, (2002).
4. Teich, J., Blickle, T., and Thiele, L.: An Evolutionary Approach to System-Level Synthesis. In *the 5th International Workshop on Hardware/Software Co-Design (Codes/CASHE '97)*, March, (1997).
5. Slomka, F., Dörfel, M., Münzenberger, R., and Hofmann, R.: Hardware/Software Codesign and Rapid Prototyping of Embedded Systems. *IEEE Design & Test of Computers*, (2000) 28-38.
6. Ljolo, M., Raghunathan, A., Dey, S., Lavagno, L., and Sangiovanni-Vincentelli, A.: Efficient Power Estimation Techniques for HW/SW systems. In *Proc. of the IEEE VOLTA'99 International Workshop on Low Power Design*, Italy, (1999) 191-199.
7. Bambha, N., Bhattacharyya, S., Teich, J., and Zitzler, E.: Hybrid Global/Local Search for Dynamic Voltage Scaling in Embedded Multiprocessor. In *Proc. of the 1st International symposium on Hardware/Software Co-design (CODES'01)*, (2001) 243-248.
8. Gruian, F.: System-Level Design Methods for Low Energy Architectures Containing Variable Voltage Processors. In *Proc. of Power-Aware Computing Systems Workshop*, November 12, Cambridge (MA), US, (2000).

9. Gruian, F., and Kuchcinski, K.: LEneS: Task-Scheduling for Low Energy Systems Using Variable Supply Voltage Processors. In *Proc. of Asia and South Pacific Design Automation Conference (ASP-DAC'01)*, (2001) 449-455.
10. Schmitz, M., Al-Hashimi, B., and Eles, P.: Synthesizing Energy-efficient Embedded Systems with LOPOCOS. *Design Automation for Embedded Systems*, 6, (2002) 401-424.
11. Mohsen, A., and Hofmann, R.: Power Modelling, Estimation, and Optimization for Automated Co-Design of Real-Time Embedded Systems. In *Proc. of the 14th International Workshop on Power and Timing Modelling, optimization and Simulation, (PATMOS'04)*, Greece, (2004) 643-651.
12. Mitschele-Thiele and Slomka, F.: *Co-design with SDL/MSC*. IT Press, (1999).
13. Pering, T., Burd, T., and Broderson, R.: The Simulation and Evaluation of Dynamic Voltage Scaling Algorithms. *ISELPED 98*, (Monterey, CA USA, August 10-12, 1998), ACM, (2000) 76-81.
14. Burd, T., and Broderson, R.: Design Issues for Dynamic Voltage Scaling. In *Proc. of the 2000 International symposium on Low power electronics and design*, Italy, (2000) 9 – 14.
15. Schmitz, M. and Al-hashimi, B.: Considering Power Variation of DVS Processing Elements for Energy Minimization in Distributed Systems. In *Proc. of the International symposium on System Synthesis (ISSS'01)*, (2001) 250-255.
16. Zitzler, E., Laumanns, M., and Thiele, L.: SPEA2: Improving the Strength Pareto Evolutionary Algorithm for Multiobjective Optimization. *Evolutionary Methods for Design, Optimization, and Control*, CIMNE, Barcelona, Spain, (2002) 95-100.
17. Topcuouglu, H., Hariri, S., and Wu, M.: Performance-effective and Low-complexity Task Scheduling for Heterogeneous Computing. *IEEE Transactions on Parallel and Distributed Systems*, Vol. 13, No. 3, (2002) 260-274.
18. The Wallenberg Laboratory for Research on Information Technology and Autonomous Systems. Available at http://www.ida.liu.se/ext/witas.
19. Dick, R., Rhodes, D., and Wolf, W. TGFF: Tasks Graphs for Free. In *Proc. of International Workshop on Hardware/Software Codesign*, March, (1998).
20. Landman, P. *Low-Power Architectural Design Methodologies*, Ph.D. Thesis, U.C. Berkeley, August (1994).

Energy-Effective Instruction Fetch Unit for Wide Issue Processors

Juan L. Aragón[1] and Alexander V. Veidenbaum[2]

[1] Dept. Ingen. y Tecnología de Computadores,
Universidad de Murcia, 30071 Murcia, Spain
jlaragon@ditec.um.es
[2] Dept. of Computer Science,
University of California, Irvine, 92697-3425 Irvine, CA, USA
alexv@cecs.uci.edu

Abstract. Continuing advances in semiconductor technology and demand for higher performance will lead to more powerful, superpipelined and wider issue processors. Instruction caches in such processors will consume a significant fraction of the on-chip energy due to very wide fetch on each cycle. This paper proposes a new energy-effective design of the fetch unit that exploits the fact that not all instructions in a given I-cache fetch line are used due to taken branches. A *Fetch Mask Determination unit* is proposed to detect which instructions in an I-cache access will actually be used to avoid fetching any of the other instructions. The solution is evaluated for a 4-, 8- and 16-wide issue processor in 100nm technology. Results show an average improvement in the I-cache Energy-Delay product of 20% for the 8-wide issue processor and 33% for the 16-wide issue processor for the SPEC2000, with no negative impact on performance.

1 Introduction

Energy consumption has become an important concern in the design of modern high performance and embedded processors. In particular, I- and D-caches and TLBs consume a significant portion of the overall energy. For instance, the I-cache energy consumption was reported to be 27% of the total energy in the StrongArm SA110 [17]. Combined I- and D-cache energy consumption accounts for 15% of the total energy in the Alpha 21264 [9]. In addition, continuing advances in semiconductor technology will lead to increased transistor count in the on-chip caches, and the fraction of the total chip energy consumed by caches is likely to go up. Other design trends such as wider issue, in case of high performance processors, or highly associative CAM-based cache organizations, commonly used in embedded processors [6][17][27], increase the fraction of energy consumed by caches. This is especially true for the I-cache, which is accessed almost every cycle. For these reasons, the energy consumption of the fetch unit and the I-cache is a very important concern in low-power processor design.

Several techniques have been proposed to reduce the energy consumption of TLBs [7] and caches in general. Many of them proposed alternative organizations, such as filter caches [13], way-prediction [11][20][24], way determination [18], way-

T. Srikanthan et al. (Eds.): ACSAC 2005, LNCS 3740, pp. 15 – 27, 2005.

memoization [15], cached load/store queue [19], victim caches [16], sub-banking [8][23], multiple line buffers and bitline segmentation [8], the use of small energy-efficient buffers [4][13], word-line segmentation [22], divided word-lines [26], as well as other circuit design techniques that are applicable to SRAM components. Some of these proposals provide an ability to access just a portion of the entire cache line (e.g. subbanking, wordline segmentation and divided wordlines), which is particularly useful when accessing the D-cache to retrieve a single word. The I-cache fetch is much wider and typically involves fetching an entire line. However, because of the high frequency of branches in applications, in particular taken branches, not all instructions in an I-cache line may actually be used.

The goal of this research is to identify such unused instructions and based on that to propose an energy-efficient fetch unit design for future wide issue processors. When a N-wide issue processor accesses the I-cache to retrieve N instructions from a line, not all N instructions may be used. This happens in two cases, which are depicted in Fig. 1:

1. One of the N instructions is a conditional or an unconditional branch that is taken – a *branch out* case. All instructions in the cache line after the taken branch will not be used.
2. An I-cache line contains a branch target, which is not at the beginning of the N-word line – a *branch into* case. The instructions before the target will not be used.

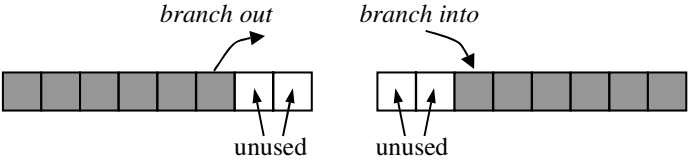


Fig. 1. *Branch out* and *branch into* cases

In this work, *Fetch Mask Determination* (*FMD*) is proposed as a technique to identify which instructions in the next I-cache line to be fetched are going to be used. Based on this information, only the useful part of the cache line is read out. Determining the unused words requires identifying the two branch cases described above. For the *branch into* case, a standard *Branch Target Buffer* (*BTB*) can be used to obtain a word address of the first useful instruction in a *next* line. For the *branch out* case, a different table is used to track if the next line to be fetched contains a conditional branch that *will* be taken. Finally, both *branch into* and *branch out* cases may occur in the same line. Therefore, both cases are combined to identify all instructions to be fetched in a given access. Once the useful instructions have been identified, the I-cache needs the ability to perform a partial access to an I-cache line. This may be achieved by using either a subbanked [8][23], wordline segmentation [22] or a divided wordline (DWL) [26] I-cache organization. Therefore, this research assumes one of these I-cache organizations to be used. The mechanism proposed in this paper will supply a bit vector to control the corresponding subbanks, pass transistors and drivers, of the underlying I-cache type.

The rest of the paper is organized as follows. Section 2 presents the related work. Section 3 motivates and describes the proposed *Fetch Mask Determination* unit. Section 4 presents the energy efficiency of the proposed mechanism. Finally, Section 5 summarizes the main conclusions of this research.

2 Background and Related Work

There have been many hardware and architectural proposals for reducing the energy consumption of caches in general (some cited above) and in particular the energy consumption of the I-cache. In a typical SRAM-based cache organization, an access to the I-cache goes through the following steps. A decoder first decodes an address and selects the appropriate RAM row by driving one wordline in the data array and one wordline in the tag array. Along the selected row, each memory cell is associated with a pair of bitlines. Initially, the bitlines are precharged high and one of them is pulled down depending on the value stored in the memory cell. Finally, a set of sense amplifiers monitors the pairs of bitlines detecting when one changes and determining the content in the memory cell. In this organization, an entire cache line is *always* read out even if only some of the instructions are used. Several approaches, that allow a partial access of a cache line, have already been applied to D-caches, supporting the idea of selectively fetching only the desired instruction words from the I-cache.

A subbanked I-cache organization divides the cache into subbanks [8][23] and activates only the required subbanks. A subbank consists of a number of consecutive bit columns of the data array. In the I-cache case, the subbank will be equal to the width of an individual instruction, typically 32 bits wide. Such a cache has been implemented in IBM's RS/6000 [1]. The instruction cache was organized as 4 separate arrays, each of which could use a different row address.

Divided wordline (DWL) [26] and wordline segmentation [22] are used to reduce the length of a wordline and thus its capacitance. This design has been implemented in actual RAMs. It typically refers to a hierarchical address decoding and wordline driving. In a way, it is or can be made similar to subbanking.

A related approach is bitline segmentation [8], which divides a bitline using pass transistors and allows sensing of only one of the segments. It isolates the sense amplifier from all other segments allowing for a more energy efficient sensing.

A number of other techniques have also been proposed to reduce the I-cache energy consumption. Way-prediction predicts a cache way and accesses it as a direct-mapped organization [11][20][24]. A phased cache [10] separates tag and data array access into two phases. First, all the tags in a set are examined in parallel but no data access occurs. Next, if there is a hit, the data access is performed for the hit way. This reduces energy consumption but doubles a cache hit time. Way-memoization [15] is an alternative to way-prediction that stores precomputed in-cache *links* to next fetch locations aimed to bypass the I-cache tag lookup and thus, reducing tag array lookup energy. Other proposals place small energy-efficient buffers in front of caches to filter traffic to the cache. Examples include block buffers [4][23], multiple line buffers [8], the filter cache [13] and the victim cache [16]. Again, these proposals trade performance for power since they usually increase the cache hit time.

A Trace Cache [21] *could* produce a similar behavior to the *Fetch Mask Determination* unit proposed in this work. A Trace Cache line identifies a dynamic stream of instructions in execution order that are going to be executed, eliminating branches between basic blocks. Therefore, the unused instructions due to taken branches are eliminated from the trace dynamically. However, a Trace Cache introduces some other inefficiencies, such as basic block replication and a higher power dissipation, trading power for performance. An energy-efficiency evaluation of the Trace Cache is out of the scope of this paper and is part of future work.

3 Energy-Effective Fetch Unit Design

3.1 Quantitative Analysis of Unused Fetched Instructions

In this section, the number of instructions that need not be fetched per I-cache line is studied to understand how these extra accesses may impact the energy consumption of a wide-issue processor. The SPEC2000 benchmark suite was studied in a processor with issue widths of 4, 8 and 16 instructions using a 32 KB, 4-way I-cache with a line size equal to fetch width.

The baseline configuration uses a 32 KB *2-level* branch predictor (in particular a PAs branch predictor, using the nomenclature from [25], whose first level is indexed by branch PC) and assumes a fetch unit that uses a standard *prefetch buffer* organized as a queue. The purpose of the prefetch buffer is to decouple the I-cache from the decode unit and the rest of the pipeline, as well as to provide a smooth flow of instructions to the decoders even in the presence of I-cache misses. Instructions are retrieved from the I-cache one line at a time and then placed in the fetch buffer, as long as there is enough space in the buffer to accommodate the entire line.

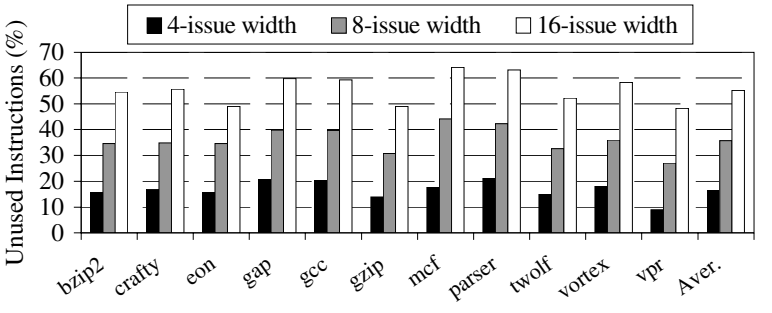


Fig. 2. Unused instructions per I-cache line for SPECint2000

Fig. 2 and Fig. 3 show the results for integer and floating point applications, respectively (see Section 4.1 for details about simulation methodology and processor configuration). For integer applications, an average of 16% of all fetched instructions are not used for the issue width of 4. This amount increases to 36% and 55% when the

issue width is increased to 8 and 16 respectively. These results show that for wide-issue processors (8-wide and up) the presence of taken branches interrupting the sequential flow of instructions is very significant and, consequently, there is a significant impact on the energy consumption of the I-cache.

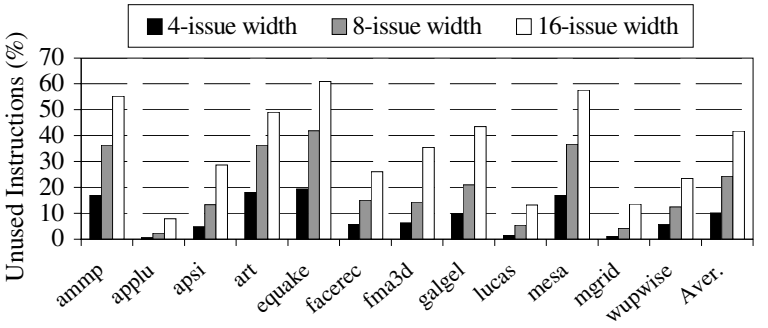


Fig. 3. Unused instructions per I-cache line for SPECfp2000

For the floating point applications (Fig. 3), the impact is not as significant. For the 8-wide issue processor the average number of unused instructions is 24%. In some benchmarks, such as *applu*, *lucas* or *mgrid*, it is less than 5%. The reason is that these applications have a high average number of instructions between branches: 200, 61, and 86, respectively. Therefore, fewer opportunities exist to locate unused instructions due to taken branches than in integer applications. For the 4-wide issue processor the average number of unused instructions is only 10%, whereas in the 16-wide issue the average is 41%, which is very significant.

These results show the potential of the proposed *Fetch Mask Determination* unit to reduce the energy consumption of the I-cache due to unused instructions in a cache line as the issue width is increased.

3.2 Fetch Mask Determination (*FMD*)

The *Fetch Mask Determination* (*FMD*) unit generates a control bit vector used to decide which instructions within an I-cache line should be fetched in the *next* cycle. The control vector is a bit mask whose length is equal to the number of instructions in a cache line. Each bit in the mask controls either the subbanks to be activated or the drivers in the segmented wordline, depending on the underlying I-cache organization, in order to access only the useful instructions for the next fetch cycle and, therefore, save energy.

To determine the bit mask for the next fetch cycle, let us consider each of the two cases described above: *branching into* and *branching out* of a cache line.

For *branching into* the next line, it is only necessary to determine whether the current fetch line contains a branch instruction that is going to be taken. This information is provided by both the *Branch Target Buffer* (*BTB*) and the conditional

branch predictor. Once a branch is predicted taken and the target address is known, its target position in the next cache line is easily determined. For this case, only instructions from the target position until the end of the cache line should be fetched. This mask is called a *target mask*.

For *branching out* of the next line, it is necessary to determine if the next I-cache line contains a branch instruction that is going to be taken. In that case, instructions from the branch position to the end of the line do not need to be fetched in the next cycle. To accomplish this, a *Mask Table* (*MT*) is used which identifies those I-cache lines that contain a branch that will be predicted as taken for its next execution. The number of entries in the *MT* equals the number of cache lines in the I-cache. Each entry of the *Mask Table* stores a binary-encoded mask, so each entry has $log_2(issue\_width)$ bits. Every cycle the *Mask Table* is accessed to determine whether the next I-cache line contains a taken branch. This mask is called a *mask of predictions*. When a branch is committed and the prediction tables are updated, we can check what the *next* prediction for this branch will be by looking at the saturating counter. This information is used to also update the *MT* in order to reflect if the branch will be predicted as taken the next time. Therefore, there are no extra accesses to the branch prediction tables, and thus, no additional power dissipation.

It is also important to note that there is no performance degradation associated with the proposed *Fetch Mask Determination* unit since it just anticipates the behavior of the underlying branch predictor to detect future taken branches, either correctly predicted or mispredicted. When that branch is executed again, the corresponding entry in *MT* will provide the correct mask for a *branch out* case, always in agreement with the branch prediction. In this way, the proposed *Fetch Mask Determination* unit is not performing any additional predictions and, therefore, it cannot miss. The *FMD* unit just uses the next prediction for a particular branch (*n* cycles before the next dynamic instance of the branch) to identify a *branch out* case or the information from the *BTB* to identify a *branch into* case. In addition, neither the I-cache hit rate nor the accuracy of the branch predictor affects the energy savings provided by our proposal, only the amount of branches predicted as taken. In case of branch misprediction, all necessary recovery actions will be done as usual and the corresponding *MT* entry will be reset to a mask of all 1's as explained below.

Finally, it is possible for both *branching into* and *branching out* cases to occur in the same cache line. In this case, the *target mask* and the *mask of predictions* need to be combined to determine which instructions to fetch in the next cycle. In order to simplify the *FMD* unit, the number of taken braches per cycle is limited to one. To better understand the proposed design, let us consider the following example for two different I-cache lines:

I-cache line1: I_1, $branch_1\_to\_target_A$, I_2, I_3
I-cache line2: I_4, $target_A$, $branch_2$, I_5

where I_j (j=1..5) and $target_A$ are non-branching instructions. Assume that *line1* is the line currently being fetched and that the branch in *line1* is predicted to be taken and its target is $target_A$ in *line2*. For this *branch into* case, *target_mask* = 0111 for the

second cache line. If $branch_2$ from *line2* is also going to be predicted as taken[1], then only the first three instructions from $line_2$ must be fetched. For this *branch out* case, the corresponding *MT* entry will provide a *mask_of_predictions* = 1110. When both masks are combined by a logical AND operation, the final mask is *next_fetch_mask* = 0110. This mask will be used for fetching just the required instructions from *line2*.

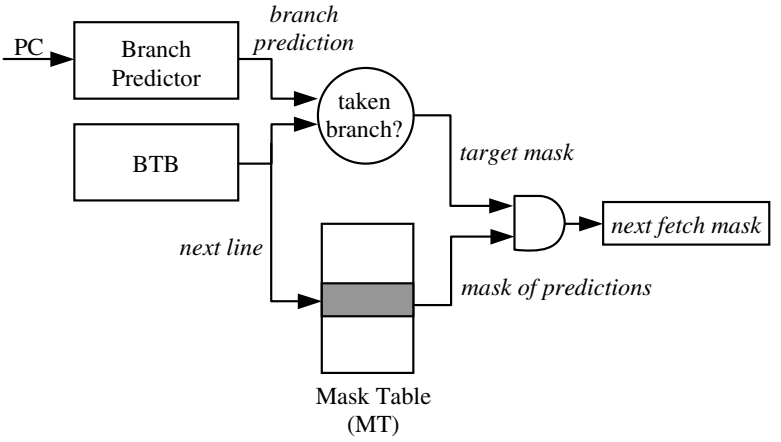


Fig. 4. The *Fetch Mask Determination* unit

The *Fetch Mask Determination* unit, depicted in Fig. 4, operates as follows:

1. Each entry in the *MT* is initialized to a mask of all 1's, which means that all instructions in the line are going to be fetched.
2. When an I-cache miss occurs and a line is replaced, the associated mask in the *MT* is reset to all 1's.
3. In the fetch stage:

```
1)  if taken branch in current line
2)  then use branch predictor/BTB to compute target_mask
3)  else target_mask = all 1's;
4)  mask_of_predictions = MT[next_line];
5)  next_fetch_mask = target_mask AND mask_of_predictions;
6)  if next_fetch_mask == 0
7)  then next_fetch_mask = target_mask;
```

The last test above (line 6) is necessary for the following case:

I-cache line1: I_1, $branch_1$_ to_ $target_A$, I_2, I_3
I-cache line2: $branch_2$, $target_A$, I_4, I_5

If the first line is fetched and $branch_1$ is predicted as not taken, the program will continue with *line2*. If $branch_2$ is taken, the *MT* will contain for *line2* a *mask_of_predictions* = 1000. The second time that *line1* is fetched, if $branch_1$ is

[1] We are assuming that its last execution changed the 2-bit saturated counter to the *predict-as-taken* state. That information was used to update the *MT* entry with mask = 1110.

taken then *target_mask* = 0111 for *line2*. The combination of both masks will result in zero, which is incorrect. According to steps 6 and 7 above, the *next fetch mask* used for fetching *line2* must be equal to *target mask*, which is the correct mask to use.

4. When updating the branch predictor at commit, also update the *MT* entry that contains the branch. If the branch being updated will be predicted as taken for the next execution, then disable all the bits from the position of the branch to the end of the line. Otherwise, set all bits to 1. Note that the update of the *MT* is performed only if the line containing the branch is still present in the I-cache.
5. In case of a branch misprediction, reset the corresponding entry in the *MT* to all 1's. There is no other effect for our proposal in case of misprediction.

As for the effect on cycle time, note that determining the *next fetch mask* for cycle *i+1* is a two-step process. In cycle *i* the *BTB* is used to create a *target mask* for the next line. Then, the next line PC is used to access the *MT* to determine the *mask of predictions* and finally, both masks are ANDed. Since the *BTB* and *MT* accesses are sequential, the *next fetch mask* may not be ready before the end of cycle *i*. If this is the case, despite a very small *MT* size (3 Kbits – see details at the end of Section 4.2), time can be borrowed from cycle *i+1* while the address decode is in progress, before accessing the data array. Note that the decode time for the data array takes about 50% of the total access time for a 32 KB, 4-way cache per CACTI v3.2 [22]. In any case, for the chosen configurations and sizes of the I-cache, *BTB* and *MT* (shown in Table 2 in Section 4.1), the sequential access time for both the *BTB* plus *MT* has been measured to be lower than the total access time of the 32 KB, 4-way I-cache (0.95 ns) as provided by CACTI.

4 Experimental Results

4.1 Simulation Methodology

To evaluate the energy efficiency of the proposed *FMD* unit, the entire SPEC2000 suite was used[2]. All benchmarks were compiled with highest optimization level (`-O4 -fast`) by the Compaq Alpha compiler and were run using a modified version of the Wattch v1.02 power-performance simulator [5]. Due to the large number of dynamic instructions in some benchmarks, we used the *test* input data set and executed benchmarks to completion. Table 1 shows, for each integer benchmark, the input set, the total number of dynamic instructions , the total number of instructions simulated, the number of skipped instructions (when necessary) and finally, the number of conditional branches.

Table 2 shows the configuration for the simulated 8-wide issue processor. The pipeline has been lengthened to 14 stages (from fetch to commit), following the pipeline of the IBM Power4 processor [14]. For the 4- and 16-wide issue processors, the L1-cache line width, reorder buffer, load-store queue, and other functional units were resized accordingly.

[2] A preliminary evaluation of an energy-efficient fetch unit design applied to embedded processors with highly associative CAM-based I-caches can be found in [3].

Table 1. SPECint2000 benchmark characteristics

Benchmark	Input set	Total # dyn. instr. input set (Mill.)	Total # simulated instr. (Mill.)	# skipped instr (Mill.)	# dyn.cond. branch (Mill.)
bzip2	input source 1	2069	500	500	43
crafty	test (modified)	437	437	-	38
eon	kajiya image	454	454	-	29
gap	test (modified)	565	500	50	56
gcc	test (modified)	567	500	50	62
gzip	input.log 1	593	500	50	52
mcf	test	259	259	-	31
parser	test (modified)	784	500	200	64
twolf	test	258	258	-	21
vortex	test (modified)	605	500	50	51
vpr	test	692	500	100	45

Table 2. Configuration of the 8-wide issue processor. For simplicity, only *one* taken branch is allowed per cycle.

Fetch engine	Up to 8 instr/cycle, 1 taken branch, 2 cycles of misprediction penalty.
BTB	1024 entries, 2-way
Branch Predictor	32 KB PAs branch predictor (2-level)
Execution engine	Issues up to 8 instr/cycle, 128-entry ROB, 64-entry LSQ.
Functional Units	8 integer alu, 2 integer mult, 2 memports, 8 FP alu, 1 FP mult.
L1 Instr-cache	32 KB, 4-way, 32 bytes/line, 1 cycle hit lat.
L1 Data-cache	64 KB, 4-way, 32 bytes/line, 3 cycle hit lat.
L2 unified cache	512 KB, 4-way, 64 b/line, 12 cycles hit lat.
Memory	8 bytes/line, 120 cycles latency.
TLB	128 entries, fully associative.
Technology	0.10µm, Vdd = 1.1 V, 3000 MHz.

4.2 Cache Energy Consumption Model

To measure the energy savings of our proposal, the Wattch simulator was augmented with a power model for the *FMD* unit. Since the original Wattch power model was based on CACTI version 1, the dynamic power model has been changed to the one from CACTI version 3.2 [22] in order to increase its accuracy. It assumed an aggressive clock gating technique: unused structures still dissipate 10% of their peak dynamic power. The power model was extended to support partial accesses to a cache line assuming a sub-banked I-cache organization.

According to [12][22], the main sources of cache energy consumption are E_{decode}, $E_{wordline}$, $E_{bitline}$, $E_{senseamp}$ and $E_{tagarray}$. The total I-cache energy is computed as:

$$E_{cache} = E_{decode} + E_{wordline} + E_{bitline} + E_{senseamp} + E_{tagarray} \quad (1)$$

Our proposal reduces the $E_{wordline}$, $E_{bitline}$ and $E_{senseamp}$ terms since they are proportional to the number of bits fetched from the I-cache line. In general, the

$E_{wordline}$ term is very small (< 1%), whereas both $E_{bitline}$ and $E_{senseamp}$ terms account for approximately 65% of the 32 KB, 4-way I-cache energy as determined by CACTI v3.2 (which is comparable with results in [8]).

With respect to the extra power dissipated by the hardware added by the *FMD* unit, note that the size of the *MT* table is very small compared to the size of the I-cache. As cited in Section 3.2, the *MT* has the same number of entries as I-cache lines and each *MT* entry has $log_2(issue\_width)$ bits. For example, for an 8-wide issue processor with a 32 KB I-cache, the size of the *MT* is just 3 Kbits[3], which is 85 times smaller than the I-cache. In this case, the power dissipated by the *MT* has been measured to be about 1.5% of the power dissipated by the whole I-cache, which is not significant in the total processor power consumption.

4.3 Energy Efficiency of Fetch Mask Determination

This section presents an evaluation of the proposed *FMD* mechanism in a 4-, 8- and 16-wide issue processor. Figures 5 and 6 show the I-cache Energy-Delay product (EDP)[4] improvement for the SPECint2000 and SPECfp2000 respectively. In addition, the improvement achieved by an *Oracle* mechanism is also evaluated. The *Oracle* mechanism identifies precisely all the instructions used within a line in each cycle providing an upper bound on the benefits of the design proposed here.

According to the analysis in Section 3.1, integer applications should provide more energy savings than floating point applications. As expected, Fig. 5 shows an average EDP improvement of just 10% for the 4-wide issue processor in integer codes. However, for wider issue processors the improvement increases to 20% for the 8-wide issue and 33% for the 16-wide issue. Similar trends are observed in all integer applications. Some benchmarks, such as *mcf* and *parser*, show an EDP improvement of up to 28% and 43% for the 8- and 16-issue width respectively. This high improvement is achieved because they have the lowest number of instructions per branches (*IPB*). Therefore, there is an inverse correlation between the *IPB* and the benefits of the design proposed here.

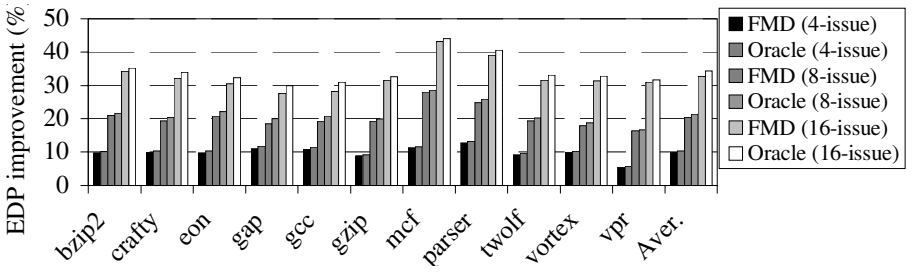


Fig. 5. I-cache Energy-Delay product improvement for SPECint2000

[3] This I-cache has 1024 lines, each containing eight 32-bit instructions. So, the *MT* size is 1024*3 bits.

[4] Energy savings are identical to the EDP product improvement since there is no performance degradation.

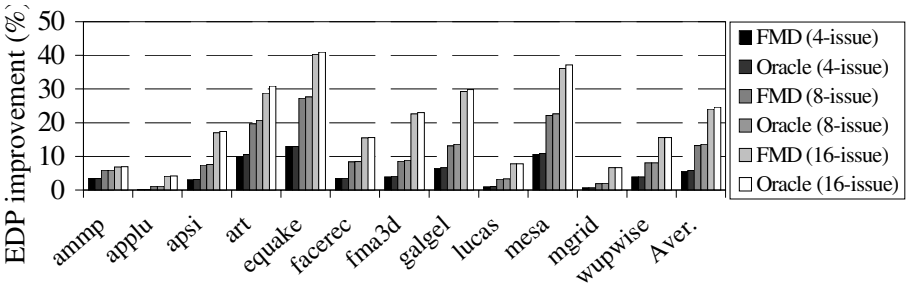


Fig. 6. I-cache EDP improvement for SPECfp2000

It is also interesting to note that *FMD* obtains an EDP improvement very close to that of the *Oracle* experiment in all benchmarks (less than 2%). This shows the effectiveness of *FMD* in determining unused instructions within an I-cache line.

For floating point applications, Fig. 6 shows an average EDP improvement of 13% for the 8-wide issue processor and 24% for the 16-wide issue processor. As in Section 3.1, some benchmarks such as *applu*, *lucas* and *mgrid* show significantly reduced EDP improvement for the 8-wide issue processor (less than 3%) due to the large number of instructions between branches. However, other floating-point applications, such as *equake* and *mesa*, have similar behavior to integer applications, and therefore, a similar EDP improvement: 40% and 36% respectively for the 16-wide issue processor.

In summary, the proposed *FMD* unit is able to provide a significant I-cache energy-delay product improvement, by not reading out of the data array in the I-cache instructions that will not be used due to taken branches. Its performance is very close to the optimal case for all benchmarks.

5 Conclusions

A modern superscalar processor fetches, but may not use, a large fraction of instructions in an I-cache line due to the high frequency of taken branches. An energy-efficient fetch unit design for wide issue processors has been proposed by means of *Fetch Mask Determination* (*FMD*), a technique able to detect such unused instructions with no performance degradation. The proposed *FMD* unit provides a bit vector to control the access to the required subbanks or to control the pass transistors in case of a segmented wordline I-cache organization. It has no impact on execution time.

The proposed design was evaluated for 4-, 8- and 16-wide issue processors. Results show an average improvement in I-cache Energy-Delay product of 10% for a 4-wide issue processor, 20% for an 8-wide issue processor and 33% for a 16-wide issue processor in integer codes. Some floating point applications show a lower EDP improvement because of the large number of instructions between branches. In addition, the proposed design was proven to be very effective in determining unused instructions in an I-cache line, providing an EDP improvement very close (< 2%) to the optimal case for all benchmarks.

Finally, the *FMD* unit is a mechanism orthogonal to other energy-effective techniques, such as fetch gating/throttling mechanisms [2] and/or way-prediction, and it can be used in conjunction with such techniques providing further energy savings.

Acknowledgements

This work has been partially supported by the Center for Embedded Computer Systems at the University of California, Irvine and by the Ministry of Education and Science of Spain under grant TIC2003-08154-C06-03. Dr. Aragón was also supported by a post-doctoral fellowship from Fundación Séneca, Región de Murcia (Spain).

References

1. IBM RISC System/6000 Processor Architecture Manual.
2. J.L. Aragón, J. González and A. González. "Power-Aware Control Speculation through Selective Throttling". *Proc. Int. Symp. on High Performance Computer Architecture (HPCA'03)*, Feb. 2003.
3. J.L. Aragón, D. Nicolaescu, A. Veidenbaum and A.M. Badulescu. "Energy–Efficient Design for Highly Associative Instruction Caches in Next–Generation Embedded Processors". *Proc. of the Int. Conference on Design, Automation and Test in Europe (DATE'04)*, Feb. 2004.
4. I. Bahar, G. Albera and S. Manne. "Power and Performance Trade-Offs Using Various Caching Strategies". *Proc. of the Int. Symp. on Low-Power Electronics and Design*, 1998
5. D. Brooks, V. Tiwari, and M. Martonosi. "Wattch: A Frame-Work for Architectural-Level Power Analysis and Optimizations". *Proc. of the Int. Symp. on Computer Architecture*, 2000.
6. L.T. Clark *et al*. "An embedded 32b microprocessor core for low-power and high-performance applications". *IEEE Journal of Solid State Circuits*, 36(11), Nov. 2001.
7. L.T. Clark, B. Choi, and M. Wilkerson. "Reducing Translation Lookaside Buffer Active Power". *Proc. of the Int. Symp. on Low Power Electronics and Design*, 2003.
8. K. Ghose and M.B. Kamble. "Reducing Power in Superscalar Processor Caches using Subbanking, Multiple Line Buffers and Bit-line Segmentation". *Proc. Int. Symp. on Low Power Electronics and Design*, pp 70-75, 1999.
9. M.K. Gowan, L.L. Biro and D.B. Jackson. "Power Considerations in the Design of the Alpha 21264 Microprocessor". *Proc. of the Design Automation Conference*, June 1998.
10. A. Hasegawa et al. "SH3: High Code Density, Low Power". *IEEE Micro*, 15(6):11-19, December 1995.
11. K. Inoue, T. Ishihara, and K. Murakami. "Way-Predicting Set-Associative Cache for High Performance and Low Energy Consumption". *Proc. Int. Symp. on Low Power Electronics and Design*, pp 273-275, Aug. 1999.
12. M. B. Kamble and K. Ghose. "Analytical Energy Dissipation Models for Low Power Caches". *Proc. Int. Symp. on Low-Power Electronics and Design*, August 1997.
13. J. Kin, M. Gupta and W.H. Mangione-Smith. "The Filter Cache: An Energy Efficient Memory Structure". *Proc. Int. Symp. on Microarchitecture*, December 1997.
14. K. Krewell. "IBM's Power4 Unveiling Continues". *Microprocessor Report*, Nov. 2000.
15. A. Ma, M. Zhang and K. Asanovic. "Way Memoization to Reduce Fetch Energy in Instruction Caches", ISCA Workshop on Complexity-Effective Design, July 2001.

16. G. Memik, G. Reinman and W.H. Mangione-Smith. "Reducing Energy and Delay using Efficient Victim Caches". *Proc. Int. Symp. on Low Power Electronics and Design*, 2003.
17. J. Montanaro *et al.* "A 160Mhz, 32b, 0.5W CMOS RISC Microprocessor". *IEEE Journal of Solid State Circuits*, 31(11), pp 1703-1712, November 1996.
18. D. Nicolaescu, A.V. Veidenbaum and A. Nicolau. "Reducing Power Consumption for High-Associativity Data Caches in Embedded Processors". *Proc. Int. Conf. on Design, Automation and Test in Europe (DATE'03)*, pp. 11064-11069, March 2003.
19. D. Nicolaescu, A.V. Veidenbaum and A. Nicolau. "Reducing Data Cache Energy Consumption via Cached Load/Store Queue". *Proc. Int. Symp. on Low Power Electronics and Design (ISLPED'03)*, pp. 252-257, August 2003.
20. M. D. Powell, A. Agarwal, T. Vijaykumar, B. Falsafi, and K. Roy. "Reducing Set-Associative Cache Energy via Way-Prediction and Selective Direct-Mapping". *Proc. Int. Symp. on Microarchitecture*, December 2001.
21. E. Rotenberg, S. Bennett, and J.E. Smith. "Trace Cache: A Low Latency Approach to High Bandwidth Instruction Fetching". *Proc. of the 29th Int. Symp. on Microarchitecture*, Nov. 1996.
22. P. Shivakumar and N.P. Jouppi. "Cacti 3.0: An Integrated Cache Timing, Power and Area Model". Tech. Report 2001/2, Digital Western Research Lab., 2001.
23. C. Su and A. Despain. "Cache Design Tradeoffs for Power and Performance Optimization: A Case Study". *Proc Int. Symp. on Low Power Design*, 1995.
24. W. Tang, A.V. Veidenbaum, A. Nicolau and R. Gupta. "Integrated I-cache Way Predictor and Branch Target Buffer to Reduce Energy Consumption". *Proc. of the Int. Symp. on High Performance Computing, Springer LNCS 2327*, pp.120-132, May 2002.
25. T.Y. Yeh and Y.N. Patt. "A Comparison of Dynamic Branch Predictors that Use Two Levels of Branch History". *Proc. of the Int. Symp. on Computer Architecture*, pp. 257-266, 1993.
26. M. Yoshimito, K. Anami, H. Shinohara, T. Yoshihara, H. Takagi, *et al*. "A Divided Word-Line Structure in the Static RAM and its Application to a 64k Full CMOS RAM". *IEEE J. Solid-State Circuits*, vol. SC-18, pp. 479-485, Oct. 1983.
27. M. Zhang and K. Asanovic. "Highly-Associative Caches for Low-power processors". *Proc. Kool Chips Workshop, 33rd Int. Symp. on Microarchitecture*, 2000.

Rule-Based Power-Balanced VLIW Instruction Scheduling with Uncertainty

Shu Xiao, Edmund M.-K. Lai, and A.B. Premkumar

School of Computer Engineering,
Nanyang Technological University, Singapore 639798
shu_x@pmail.ntu.edu.sg,
{asmklai, asannamalai}@ntu.edu.sg

Abstract. Power-balanced instruction scheduling for Very Long Instruction Word (VLIW) processors is an optimization problem which requires a good instruction-level power model for the target processor. Conventionally, these power models are deterministic. However, in reality, there will always be some degree of imprecision involved. For power critical applications, it is desirable to find an optimal schedule which makes sure that the effects of these uncertainties could be minimized. The scheduling algorithm has to be computationally efficient in order to be practical for use in compilers. In this paper, we propose a rule based genetic algorithm to efficiently solve the optimization problem of power-balanced VLIW instruction scheduling with uncertainties in the power consumption model. We theoretically prove our rule-based genetic algorithm can produce as good optimal schedules as the existing algorithms proposed for this problem. Furthermore, its computational efficiency is significantly improved.

1 Introduction

Power-balanced instruction scheduling for very long instruction word (VLIW) processors is the task of producing a schedule of VLIW instructions so that the power variation over the execution time of the program is minimized, while the deadline constraints are met. Most currently instruction scheduling techniques for this problem are based on deterministic power models [1, 2, 3]. Since these instruction level models are estimated from empirical measurements [4, 5], there will always be some degree of imprecision or uncertainty. Furthermore, in order to reduce the complexity of the power model, some approximation techniques such as instruction clustering [6] have to be employed which contributes to the imprecision involved. While these instruction scheduling techniques using the approximate deterministic power models allow us to optimize power consumption in the average sense, it is desirable to find an optimal schedule which ensures that the effects of those uncertainties could be minimized for power critical applications.

A rough programming approach has previously been proposed to solve this problem [7, 8, 9]. This approach is shown in Fig. 1. An instruction-level power

T. Srikanthan et al. (Eds.): ACSAC 2005, LNCS 3740, pp. 28–40, 2005.

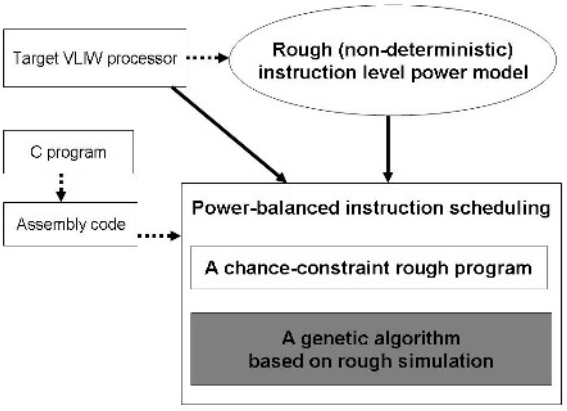


Fig. 1. A rough programming approach proposed in [7,8,9]

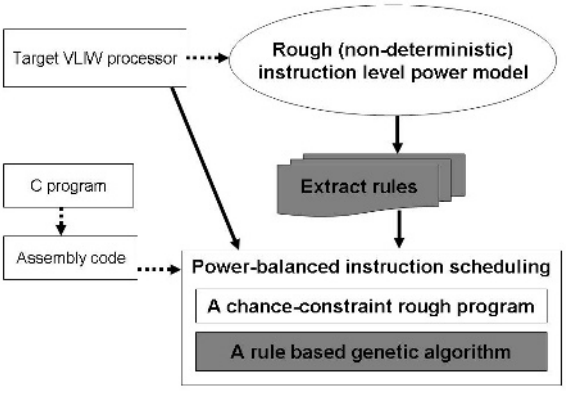


Fig. 2. A rule based genetic algorithm to solve the chance-constraint rough program

modelling technique which applies rough set theory is used to handle the uncertainties involved [7,8]. Then the power-balanced instruction scheduling problem based on the rough set based power model is formulated as a chance-constraint rough program [7,9]. This program was previously solved by a genetic algorithm (GA) [9]. In order to rank the schedules produced in each generation of the genetic algorithm, rough simulation is used [10]. It is a simulation process that estimates the ranges of the objective function values for a given schedule given a confidence level. It is very expensive computationally. Thus while this technique is of interest academically it is not suitable for practical use.

In this paper, we propose a rule-based genetic algorithm to solve the above optimization problem much more efficiently. The steps involved in this new method is summarized in Fig. 2. It ranks the generated schedules by simply comparing

the differences between the objective function values of these schedules using a set of rules instead of rough simulation. These rules are generated by learning the decision making process of rough simulations. This rule extraction process, though computationally expensive since it involves rough simulation, only needs to be performed once off-line for a particular target processor. Thus the computation time involved in the instruction scheduling phase is much reduced.

We shall review the rough power model, the chance-constraint rough program formulation of the problem and the existing GA to solve it in Sections 2, 3 and 4 respectively. In Section 5.1, our rule-based GA is presented. We proved mathematically that the solutions obtained by the rule-based GA are as good as those obtained using the existing GA. The rule extraction method is discussed in Section 5.2. Then examples are given to illustrate how the improved GA using these rules is able to improve the computational efficiency substantially.

2 Rough Instruction Level Power Model

Suppose a program schedule X consists of T time slots:

$$X =< w_1, ..., w_{i-1}, w_i, ..., w_T >$$

where w_i is the long instruction word in the i-th time slot. A most common approach to estimate the power consumed in time slot i during the execution of X is

$$P_i \approx U(0|0) + \sum_{k=1}^{F} v^k_{(i|i-1)} \tag{1}$$

Here, $U(0|0)$ is the base power cost which is the power consumed by the execution of an instruction word constituted entirely by no-operation (NOP) instructions. F is the number of functional units in the target VLIW processor. the summations of $v^k_{(i|i-1)}$ is the additional power contributions on the F functional units due to the change of instructions on the same functional unit in the time slot i. The number of instruction pairs to be considered for $v^k_{(i|i-1)}$ in (1) could become too large to be characterized, since two instructions differ either in terms of functionality (i.e., opcode), addressing mode (immediate, register, indirect, etc.), or data differences (either in terms of register names or immediate values).

The complexity can be reduced by instruction clustering, that is instructions are categorized into classes, such that the instructions in a given class are characterized by very similar power cost. Then a simplified model from (1) can be obtained:

$$P_i \approx U(0|0) + \sum_{k=1}^{C} r_k c_k \tag{2}$$

where the additional power consumption to $U(0|0)$ is computed as the summations of the power due to the being executed instructions in different clusters in

the time slot i. C is the number of instruction clusters. c_k is the power consumption parameter representing power consumption of instructions in the cluster k. r_k represents the number of being executed instruction belong to the cluster k in the time slot i.

However, simply by instruction clustering each c_k only represents an average power consumption for the instructions in the same cluster with different opcodes, addressing modes, operands or the preceding opcodes. In order to indicate the uncertainty involved in each power consumption parameter, each c_k is expressed as a rough variable represented by $([a,b],[c,d])$. $[a,b]$ is its lower approximation and $[c,d]$ its upper approximation and $c \leq a \leq b \leq d$ are real numbers [8]. This means that the values within $[a,b]$ are sure and those within $[c,d]$ are possible.

3 Chance-Constraint Rough Program Formulation

The total power deviation for a schedule X is proportional to the power squared and is given by

$$PV(X) = \sum_{i=1}^{T} (P_i - M)^2 \tag{3}$$

where the average power over T time slots is given by

$$M = \left(\sum_{i=1}^{T} P_i \right) / T \tag{4}$$

This is the function we seek to minimize.

If each c_k in (2) is represented as a rough variable, then the return values of P_i in (2), M in (4) and $PV(X)$ in (3) are also rough. They can be ranked by their α-pessimistic values for some predetermined confidence level $\alpha \in (0,1]$. Our optimization problem needs to make sure that the effects of the uncertainties could be minimized. Thus a large enough confidence level is required.

Definition 1. *Let ϑ be a rough variable given as $([a,b],[c,d])$, and $\alpha \in (0,1]$. Then*

$$\vartheta_{\inf}^{\alpha} = \inf \{r | Tr\{\vartheta \leq r\} \geq \alpha\} \tag{5}$$

is called the α-pessimistic value of ϑ, where Tr is the trust measure operator, defined as

$$Tr\{\vartheta \leq r\} = \begin{cases} 0, & r \leq c \\ \frac{1}{2}\left(\frac{c-r}{c-d}\right), & c \leq r \leq a \\ \frac{1}{2}\left(\frac{c-r}{c-d} + \frac{a-r}{a-b}\right), & a \leq r \leq b \\ \frac{1}{2}\left(\frac{c-r}{c-d} + 1\right), & b \leq r \leq d \\ 1, & r \geq d \end{cases} \tag{6}$$

Combining (5) and (6), we have

$$\vartheta_{\inf}^{\alpha} = \begin{cases} (1-2\alpha)c + 2\alpha d, & 0 < \alpha \leq \frac{a-c}{2(d-c)} \\ 2(1-\alpha)c + (2\alpha-1)d, & \frac{b+d-2c}{2(d-c)} \leq \alpha \leq 1 \\ \frac{c(b-a)+a(d-c)+2\alpha(b-a)(d-c)}{(b-a)+(d-c)}, & \frac{a-c}{2(d-c)} < \alpha < \frac{b+d-2c}{2(d-c)} \end{cases} \tag{7}$$

The chance-constraint rough program is given by $\mathbf{P_{rp}}$.

$\mathbf{P_{rp}}$: $\min \ PV(X,\xi)_{inf}^{\alpha}$

subject to

$$X = \bigcup x^j, \quad j = 1, ..., N \tag{8}$$

$$1 \leq x^j \leq T, \quad j = 1, ..., N \tag{9}$$

$$\xi = \bigcup c_k, \quad k = 1, ..., C \tag{10}$$

$$G(X) \leq 0 \tag{11}$$

$$L(X) = 0 \tag{12}$$

The objective function defined by

$$PV(X,\xi)_{inf}^{\alpha} = inf\{q | Tr\{PV(X,\xi) \leq q\} \geq \alpha\} \tag{13}$$

is based on its α-pessimistic value where α is the large enough confidence level. Let N denote total number of instructions. A schedule X can be represented by a set of integer variables x^j, which denote the allocated time slots for these instructions. ξ is the set of rough power consumption parameters defined in (2). (8), (11) and (12) define the constraint matrix for the processor-specific resource constraints, and data dependency constraints.

4 Existing GA Solution

Since the objective function of a rough program is multimodal and the search space is particularly irregular, conventional optimization techniques are unable to produce near-optimal solutions. Based on the general GA framework proposed in [10], a problem-specific GA has been proposed [9] to solve the formulation in Section 3. The three main elements of this algorithm are outlined as follows.

1. **Initial Population**: An initial population of candidate schedules is a set of feasible schedules created randomly and "seeded" with schedules obtained through conventional (non-power-aware) scheduling algorithms.
2. **Fitness Evaluation and Selection**: The objective function defined by (13) is used to evaluate the fitness of schedules in each generation. Then they are sorted non-decreasingly in terms of their fitness. In order to compute the objective function (13) of a given schedule X, the following rough simulation process is used. Let R be the sampling size. For each power consumption parameter $c_i \in \xi$ $(i = 1, 2, 3, \ldots)$, randomly take R samples from its lower and

upper approximations, $l_i^k(k=1,\ldots,R)$ and $u_i^k(k=1,\ldots,R)$, respectively. The value of the function $PV\,(X,\xi)_{\inf}^{\alpha}$ is given by the minimum value of v such that

$$\frac{l\,(v)+u\,(v)}{2R}\geq\alpha$$

where $l\,(v)$ denotes the number of $PV\left(X,l_1^k,\ldots,l_i^k,\ldots\right)_{\inf}^{\alpha}\leq v$ being satisfied when $k=1,\ldots,R$ respectively; and $u\,(v)$ denotes the number of $PV\left(X,u_1^k,\ldots,u_i^k,\ldots\right)_{\inf}^{\alpha}\leq v$ being satisfied when $k=1,\ldots,R$ respectively. The rough simulation process is summarized as in Algorithm 1.

input : A feasible schedule X; lower and upper approximations of each power consumption parameter $c_i\in\xi$; confidence level α; let R be the sampling size
output: Return of $PV(X,\xi)_{inf}^{\alpha}$

1 **for** $k\leftarrow 1$ **to** R **do**
2 | **foreach** *power consumption parameter* $c_i\in\xi$ **do** randomly sample l_i^k from its lower approximation;
3 | **foreach** *power consumption parameter* $c_i\in\xi$ **do** randomly sample u_i^k from its upper approximation;
4 **end**
5 $l\,(v)\leftarrow$ the number of $PV\left(X,l_1^k,\ldots,l_i^k,\ldots\right)_{\inf}^{\alpha}\leq v$ being satisfied when $k=1,\ldots,R$ respectively;
6 $u\,(v)\leftarrow$ the number of $PV\left(X,u_1^k,\ldots,u_i^k,\ldots\right)_{\inf}^{\alpha}\leq v$ being satisfied when $k=1,\ldots,R$ respectively;
7 Find the minimal value v such that

$$\frac{l\,(v)+u\,(v)}{2R}\geq\alpha$$

8 Return v;

Algorithm 1. Rough Simulation algorithm

3. **Crossover and Mutation**: The selected parents are divided into pairs and crossovered using 2-point crossover operator. The motivation for using mutation, then, is to prevent the permanent loss of any particular bit or allele (premature convergence).

When a pre-determined number of generations is reached, the algorithm stops. The maximum number of generations depends on the size of the problem, i.e. the number of instructions and the number of available time slots.

5 Rule-Based GA Solution

5.1 Theoretical Basis

The rough simulation algorithm provides an estimate of the range of the objective function values of a schedule by simulating the possible values for every power

consumption parameter for the whole duration of the program. Thus the larger the sample size, the better the simulation result. The lengthy computation makes this GA not practical for use in compilers.

However, we note that for fitness evaluation and selection, we don't have to compute the objective function values defined by (13) for each schedule. We actually only need to know the relative amounts. In this section, we shall establish the theoretical basis for obtaining the differences in the objective function values of a set of generated schedules. Lemma 1, Theorem 1 and Corollary 1 also prove that we can obtain the same result in fitness evaluation and selection as that given by rough simulation. By removing the time consuming rough simulation, the computational efficiency of the GA is significantly improved.

Lemma 1. *Given two rough variables* $y = ([a_y, b_y], [c_y, d_y])$ *and* $z = ([a_z, b_z], [c_z, d_z])$*, let* $u = x + y$*. Given a confidence level* $\alpha \in (0, 1]$ *which satisfies*

$$\alpha \geq \max\left(\frac{b_y + d_y - 2c_y}{2(d_y - c_y)}, \frac{b_z + d_z - 2c_z}{2(d_z - c_z)}\right)$$

we have

$$u^\alpha_{\inf} = x^\alpha_{\inf} + y^\alpha_{\inf} \tag{14}$$

Proof. Since u is the sum of x and y, the lower and upper approximations of u are computed by adding the values of the corresponding limits (see rough variable arithmetics in [10]):

$$u = ([a_y + a_z, b_y + b_z], [c_y + c_z, d_y + d_z])$$

$u^\alpha_{\inf}$, $y^\alpha_{\inf}$ and $z^\alpha_{\inf}$ are the α-pessimistic values of u, y and z respectively. Based on (7), these values are given by

$$u^\alpha_{\inf} = \begin{cases} (1-2\alpha)(c_y + c_z) + 2\alpha(d_y + d_z), & \alpha \leq \frac{a_y + a_z - c_y - c_z}{2(d_y + d_z - c_y - c_z)} \\ 2(1-\alpha)(c_y + c_z) + (2\alpha - 1)(d_y + d_z), & \alpha \geq p_u \\ \frac{(c_y + c_z)(b_y + b_z - a_y - a_z) + (a_y + a_z)(d_y + d_z - c_y - c_z)}{(b_y + b_z - a_y - a_z) + (d_y + d_z - c_y - c_z)} & \\ \quad + \frac{2\alpha(b_y + b_z - a_y - a_z)(d_y + d_z - c_y - c_z)}{(b_y + b_z - a_y - a_z) + (d_y + d_z - c_y - c_z)}, & \text{otherwise} \end{cases} \tag{15}$$

$$y^\alpha_{\inf} = \begin{cases} (1-2\alpha)c_y + 2\alpha d_y, & \alpha \leq \frac{a_y - c_y}{2(d_y - c_y)} \\ 2(1-\alpha)c_y + (2\alpha - 1)d_y, & \alpha \geq p_y \\ \frac{c_y(b_y - a_y) + a_y(d_y - c_y) + 2\alpha(b_y - a_y)(d_y - c_y)}{(b_y - a_y) + (d_y - c_y)}, & \text{otherwise} \end{cases} \tag{16}$$

$$z^\alpha_{\inf} = \begin{cases} (1-2\alpha)c_z + 2\alpha d_z, & \alpha \leq \frac{a_z - c_z}{2(d_z - c_z)} \\ 2(1-\alpha)c_z + (2\alpha - 1)d_z, & \alpha \geq p_z \\ \frac{c_z(b_z - a_z) + a_z(d_z - c_z) + 2\alpha(b_z - a_z)(d_z - c_z)}{(b_z - a_z) + (d_z - c_z)}, & \text{otherwise} \end{cases} \tag{17}$$

where

$$p_u = \frac{b_y + b_z + d_y + d_z - 2(c_y + c_z)}{2(d_y + d_z - c_y - c_z)}$$

$$p_y = \frac{b_y + d_y - 2c_y}{2(d_y - c_y)}$$
$$p_z = \frac{b_z + d_z - 2c_z}{2(d_z - c_z)}$$

Note that $p_u < \max(p_y, p_z)$. Hence if $\alpha \geq \max(p_y, p_z)$, then $\alpha \geq p_u$. In this case, we have

$$y^{\alpha}_{inf} + z^{\alpha}_{inf} = u^{\alpha}_{inf} \tag{18}$$

This completes the proof.

Remark 1. We do not need to consider the cases $\alpha \leq p_y$ and $\alpha \leq p_z$ in (16) and (17) in Lemma 1. Our optimization problem requires that the effects of the uncertainties could be minimized. Thus a large enough confidence level is needed in which case $\alpha \geq \max(p_y, p_z)$ is required. In the rest of this paper, we assume that α satisfies this condition.

Definition 2. *Consider an instruction schedule X_1. Schedule X_2 is obtained by rescheduling a single instruction from time slot i to time slot j. Then we say that instruction schedules X_1 and X_2 exhibit a 'OneMove' difference.*

If there is a 'OneMove' difference between X_1 and X_2, then the power consumption of the two schedules are exactly the same for every time slot except i and j as shown in Table 1 where c is any one of the power consumption parameters in (2).

Table 1. Symbolic power consumption values to illustrate Definition 2

time slot	...	i	...	j	...
X_1	...	$A + c$	...	B	...
X_2	...	A	...	$B + c$	...

Theorem 1. *Suppose there is a 'OneMove' difference in time slots i and j between two instruction schedules X_1 and X_2 as shown in Table 1. Let $A = a_1 + a_2 + ... + a_n$ and $B = b_1 + b_2 + ... + b_m$. Then the difference of the objective function values defined by (13) for X_1 and X_2 is given by*

$$PV(X_2, \xi)^{\alpha}_{inf} - PV(X_1, \xi)^{\alpha}_{inf} = \sum_{i=1}^{m} (2b_i c)^{\alpha}_{\text{inf}} - \sum_{i=1}^{n} (2a_i c)^{\alpha}_{\text{inf}} \tag{19}$$

Proof. The objective function defined by (13) can be computed as a sum of the contributions from time slots i and j and that of the rest of the time slots. According to Lemma 1, with a suitable α,

$$
\begin{aligned}
PV(X_1,\xi)^{\alpha}_{inf} &= \left(\sum_{k=1,k\neq i,k\neq j}^{T}(P_k-M)^2\right)^{\alpha}_{inf} + \left((P_i-M)^2+(P_j-M)^2\right)^{\alpha}_{inf} \\
&= \left(\sum_{k=1,k\neq i,k\neq j}^{T}(P_k-M)^2\right)^{\alpha}_{inf} + \left((A+c-M)^2+(B-M)^2\right)^{\alpha}_{inf} \\
PV(X_2,\xi)^{\alpha}_{inf} &= \left(\sum_{k=1,k\neq i,k\neq j}^{T}(P_k-M)^2\right)^{\alpha}_{inf} + \left((P_i-M)^2+(P_j-M)^2\right)^{\alpha}_{inf} \\
&= \left(\sum_{k=1,k\neq i,k\neq j}^{T}(P_k-M)^2\right)^{\alpha}_{inf} + \left((A-M)^2+(B+c-M)^2\right)^{\alpha}_{inf}
\end{aligned}
$$

The difference is given by

$$
\begin{aligned}
&PV(X_2,\xi)^{\alpha}_{inf} - PV(X_1,\xi)^{\alpha}_{inf} \\
&= \left((A-M)^2+(B+c-M)^2\right)^{\alpha}_{inf} - \left((A+c-M)^2+(B-M)^2\right)^{\alpha}_{inf} \\
&= (2Bc)^{\alpha}_{inf} - (2Ac)^{\alpha}_{inf} \\
&= \sum_{i=1}^{m}(2b_ic)^{\alpha}_{\inf} - \sum_{i=1}^{n}(2a_ic)^{\alpha}_{\inf}
\end{aligned}
$$

Hence proved.

Corollary 1. *Suppose two schedules X_1 and X_2 have K $(K>1)$ 'OneMove' differences. Then the difference in the objective function values of X_1 and X_2 equals to the sum of the differences caused by each of the K 'OneMoves'.*

Proof. First consider $K=2$. There are two 'OneMove' differences between X_1 and X_2. We construct an intermediate schedule X_3 with one 'OneMove' difference compared with X_1 and another 'OneMove' difference compared with X_2. Then

$$
\begin{aligned}
&PV(X_2,\xi)^{\alpha}_{inf} - PV(X_1,\xi)^{\alpha}_{inf} \\
&= PV(X_2,\xi)^{\alpha}_{inf} - PV(X_1,\xi)^{\alpha}_{inf} + PV(X_3,\xi)^{\alpha}_{inf} - PV(X_3,\xi)^{\alpha}_{inf} \\
&= (PV(X_2,\xi)^{\alpha}_{inf} - PV(X_3,\xi)^{\alpha}_{inf}) + (PV(X_3,\xi)^{\alpha}_{inf} - PV(X_1,\xi)^{\alpha}_{inf})
\end{aligned}
$$

This completes the proof for $K=2$. $K>2$ can be proved in the same way.

5.2 Rule Extraction

Corollary 1 tells that the difference between two schedules depends on the 'OneMove' differences between them. A 'OneMove' difference is characterized by two sets of α-pessimistic values $(2b_ic)^{\alpha}_{inf}$ and $(2a_ic)^{\alpha}_{inf}$ as shown in Theorem 1. Therefore we abstract the rough simulation processes that were used for computing these α-pessimistic values by a set of rules. Then by matching the 'OneMove' differences between two given schedules with our rules, the difference between their objective function returns can be obtained.

A rule corresponds to the rough simulation process of a α-pessimistic value $(2b_ic)^{\alpha}_{inf}$ (or $(2a_ic)^{\alpha}_{inf}$). Its format is as follows:

1. The premise defines a possible combination of a_ic (or b_ic) given the target VLIW processor. Suppose the instruction set of the processor is divided into C clusters, we have C^2 combinations for (c, a_i) (or (c,b_i)). Because $(2a_ic)^{\alpha}_{inf}$ and $(2ca_i)^{\alpha}_{inf}$ are equal, the reciprocal ones are excluded. Thus we totally have $\frac{1}{2}(C^2 + C)$ rules.
2. The conclusion part is the value of $(2a_ic)^{\alpha}_{inf}$ or $(2b_ic)^{\alpha}_{inf}$ obtained through rough simulation.

Example 1. Suppose the instruction set of the target VLIW processor is divided into two clusters. The rough variables representing the two associated power consumption parameters c_1 and c_2 are given in Table 2.

Table 2. Power consumption parameters for Example 1

c_1	c_2
([19.0, 20.2], [19.0, 20.7])	([22.0, 23.0], [21.5, 23.3])

The premise of the rules are all combinations of c_1 and c_2: $\{c_1, c_2\}$: (c_1, c_2), (c_1, c_1), (c_2, c_2) and (c_2, c_1). (c_2, c_1) is actually a repetition of (c_1, c_2) because $(2c_1c_2)^{\alpha}_{inf}$ and $(2c_2c_1)^{\alpha}_{inf}$ are equal. Therefore only three rules are needed.

Let $\alpha = 0.95$. We compute $(2c_1c_2)^{\alpha}_{inf}$, $(2c_1c_1)^{\alpha}_{inf}$ and $(2c_2c_2)^{\alpha}_{inf}$ by rough simulation. The three rules are summarized as a decision table shown in Table 3.

Table 3. Decision table for Example 1

	c	$a_i(b_i)$	DiffObj
1	c_1	c_2	853.0
2	c_1	c_1	744.1
3	c_2	c_2	987.9

The next example illustrates how the rules in Example 1 can be used to rank two given schedules.

Example 2. There are seven instructions $\{s_1, s_2, s_3, s_4, s_5, s_6, s_7\}$ to be scheduled in five time slots. For simplicity, let all of them be instructions with single cycle functional unit latency. Further, assume there are no dependencies among them and there are no resource usage constraints by the target VLIW processor. Suppose s_1, s_2, s_3 and s_7 belong to cluster c_1 and s_4, s_5 and s_6 belong to cluster c_2.

Two schedules X_1 and X_2 as shown in Table 4 are to be ranked according to their "fitness".

Using the power equation (2), we substitute instructions with their corresponding (symbolic) power consumption parameters for these two schedules (For simplicity $U(0|0)$ is ignored since this part is the same in every time slot.) Then we have the power data for each time slot for the two schedules as in Table 5.

Table 4. Instruction Schedules for Example 2

X_1	TimeSlot	1	2	3	4	5
	Instructions	s_1	s_2,s_3	s_4	s_5,s_6	s_7

X_2	TimeSlot	1	2	3	4	5
	Instructions	s_1	s_2	s_3,s_4	s_5	s_6,s_7

Table 5. Power consumptions for instruction schedules in Example 2

X_1	TimeSlot	1	2	3	4	5
	Power	c_1	$c_1 + c_1$	c_2	$c_2 + c_2$	c_1

X_2	TimeSlot	1	2	3	4	5
	Power	c_1	c_1	$c_1 + c_2$	c_2	$c_1 + c_2$

Comparing the power data shown in Table 5, the two schedules exhibit two 'OneMove' differences – one between slots 2 and 3 and another between slots 4 and 5. Therefore the difference in the objective function values of X_2 and X_1 depends on the four α-pessimistic values according to Corollary 1 and Theorem 1, i.e.

$$PV(X_2,\xi)^{\alpha}_{inf} - PV(X_1,\xi)^{\alpha}_{inf} = (2c_1c_2)^{\alpha}_{inf} - (2c_2c_2)^{\alpha}_{inf} + (2c_1c_2)^{\alpha}_{inf} - (2c_1c_1)^{\alpha}_{inf}$$

The values of $(2c_1c_2)^{\alpha}_{inf}$, $(2c_2c_2)^{\alpha}_{inf}$ and $(2c_1c_1)^{\alpha}_{inf}$ can be found in Table 3. Therefore,

$$PV(X_2,\xi)^{\alpha}_{inf} - PV(X_1,\xi)^{\alpha}_{inf} = 2 \times 853.0 - 987.9 - 744.1 = -26$$

Hence X_2 is worse than X_1; its power variation defined by the objective function (13) is increased by 26 compared with X_1.

5.3 Computational Efficiency of Rule-Based GA

The computational advantage of the rule-based method can be evaluated using a real VLIW processor. Our target processor is the TMS320C6711 [11] which is a VLIW digital signal processor. The instruction set of TMS320C6711 is partitioned into four clusters as in [8]. Therefore, we have ten rules for this VLIW processor to abstract the rough simulation results of the ten α-pessimistic values.

We perform power-balanced instruction scheduling on five programs taken from MediaBench [12]. For any given target instruction block, scheduling is performed by means of GA using rough simulation and the proposed rule-based approach for fitness evaluation and selection. The sample size for rough simulation, the population size and the number of generations are 50, 20 and 20 respectively. The crossover probability, mutation rate, population size and generations are same in both cases. All our computational experiments were conducted on an Intel Pentium 4 2.80GHz personal computer with 512MB RAM running under Microsoft Windows 2000.

Table 6. Computation time of GAs on benchmarks from Mediabench

Prob. Size	Source	Rough Simulation GA (sec.)	Rule-based GA (sec.)
(28,30)	epic	55.98	0.015
(37,30)	g721	73.09	0.031
(44,23)	gsm	66.53	0.015
(38,34)	jpeg	81.83	0.031
(70,58)	mpeg2	245.90	0.046

Table 6 shows the computation time required by the two GAs. For each problem instance, the problem size refers to the number of time slots and the number of instructions in the instruction block. The results show a significant reduction in computation time. The shorter time required by the rule-based GA make this approach practical for implementation is real compilers.

6 Conclusions

This paper presents our continuing research on power-balanced instruction scheduling for VLIW processors using rough set theory to model the uncertainties involved in estimating the power model of the processor. In this paper, we proposed an efficient rule-based genetic algorithm to solve the scheduling problem which has been formulated as a rough program. The rules are used to rank the schedules produced in each generation of the GA so that selection decisions can be made. Therefore the computational efficiency of this GA is significantly improved compared with those in [9, 10]. The theoretical basis of our method is derived rigorously. The short computation time required makes the rule-based approach practical for use in production compilers.

References

1. Yun, H., Kim, J.: Power-aware modulo scheduling for high-performance VLIW processors, Huntington Beach, California, USA. (2001) 40–45
2. Yang, H., Gao, G.R., Leung, C.: On achieving balanced power consumption in software pipelined loops, Grenoble, France (2002) 210–217
3. Xiao, S., Lai, E.M.K.: A branch and bound algorithm for power-aware instruction scheduling of VLIW architecture. In: Proc. Workshop on Compilers and Tools for Constrained Embedded Syst., Washington DC, USA (2004)
4. Tiwari, V., Malik, S., Wolfe, A., Lee, M.T.: Instruction level power analysis and optimization of software. (1996) 326–328
5. Gebotys, C.: Power minimization derived from architectural-usage of VLIW processors, Los Angeles, USA (2000) 308–311
6. Bona, A., Sami, M., Sciutos, D., Silvano, C., Zaccaria, V., R.Zafalon: Energy estimation and optimization of embedded VLIW processors based on instruction clustering, New Orleans, USA (2002) 886–891
7. Xiao, S., Lai, E.M.K.: A rough programming approach to power-aware vliw instruction scheduling for digital signal processors, Philadelphia, USA (2005)

8. Xiao, S., Lai, E.M.K.: A rough set approach to instruction-level power analysis of embedded VLIW processors. In: Proc. Int. Conf. on Information and Management Sciences, Kunming, China (2005)
9. Xiao, S., Lai, E.M.K.: Power-balanced VLIW instruction scheduling using rough programming. In: Proc. Int. Conf. on Information and Management Sciences, Kunming, China (2005)
10. Liu, B.: Theory and practice of uncertain programming. Physica-Verlag, Heidelberg (2002)
11. : TMS320C621x/C671x DSP two-level internal memory reference guide. Application Report SPRU609, Texas Instruments Inc. (2002)
12. Lee, C., Potkonjak, M., Mangione-Smith, W.H.: MediaBench: a tool for evaluating and synthesizing multimedia and communications systems. (1997) 330–335

An Innovative Instruction Cache for Embedded Processors*

Cheol Hong Kim[1], Sung Woo Chung[2], and Chu Shik Jhon[1]

[1] Department of Electrical Engineering and Computer Science, Seoul National University, Shilim-dong, Kwanak-gu, Seoul, Korea
{kimch, csjhon}@panda.snu.ac.kr
[2] Department of Computer Science, University of Virginia, Charlottesville, Virginia 22904, USA
schung@cs.virginia.edu

Abstract. In this paper we present a methodology to enable the design of power efficient instruction cache for embedded processors. The proposed technique, which splits the instruction cache into several small sub-caches, utilizes the locality of applications to reduce dynamic energy consumption in the instruction cache. The proposed cache reduces dynamic energy consumption by accessing only one sub-cache when a request comes into the cache. It also reduces dynamic energy consumption by eliminating the energy consumed in tag matching. In addition, we propose the technique to reduce leakage energy consumption in the proposed cache. We evaluate the design using a simulation infrastructure based on SimpleScalar and CACTI. Simulation results show that the proposed cache reduces dynamic energy by 42% – 59% and reduces leakage energy by 70% – 80%.

1 Introduction

High energy consumption in a processor reduces the battery life of embedded systems. Moreover, it requires high cooling and packaging costs. Unfortunately, as applications continue to require more computational power, energy consumption in a processor dramatically increases. For this reason, many researches have focused on the energy efficiency of cache memories to improve the energy efficiency of a processor, because caches consume a significant fraction of total processor energy [1].

In caches, energy is consumed whenever caches are accessed (Dynamic/ Leakage energy), and even when caches are idle (Leakage energy). Dynamic energy is main concern of energy in current technology. To reduce dynamic energy consumption in the cache, many researches have been examined. Filter cache trades performance for energy consumption by filtering power-costly regular cache accesses through an extremely small cache [2]. Bellas et al. proposed a technique using an additional mini cache located between the L1 instruction

* This work was supported by the Brain Korea 21 Project.

T. Srikanthan et al. (Eds.): ACSAC 2005, LNCS 3740, pp. 41–51, 2005.

cache and the CPU core, which reduces signal switching activity and dissipated energy with the help of compiler [3]. Selective-way cache provides the ability to disable a set of the ways in a set-associative cache during periods of modest cache activity to reduce energy consumption, while the full cache may remain operational for more cache-intensive periods [4].

As the number of transistors employed in a processor increases, leakage energy consumption becomes comparable to dynamic energy consumption. Reducing leakage energy in caches is especially important, because caches comprise much of a chip's transistor counts. Various techniques have been suggested to reduce leakage energy consumption in the cache. Powell et al. proposed the gated-Vdd, a circuit-level technique to gate the supply voltage and reduce leakage energy in unused memory cells [5]. They also proposed DRI (Dynamically Resizable Instruction) cache [5] that reduces leakage energy dissipation by resizing the cache size dynamically using the gated-Vdd technique. Cache decay reduces leakage energy by invalidating and "turning off" cache lines when they hold data that are not likely to be reused, based on the gated-Vdd technique [6]. Drowsy cache scheme, composed of normal mode and drowsy mode for each cache line, reduces leakage energy consumption with multi-level supply voltages [7][8].

In this paper, we focus on the energy efficiency of the L1 instruction cache (iL1). We propose a hardware technique to reduce dynamic energy consumption in the iL1 by partitioning it to several sub-caches. When a request comes into the proposed cache, only one predicted sub-cache is accessed by utilizing the locality of applications. In the meantime, the other sub-caches are not accessed, which leads to dynamic energy reduction. We also propose the technique to reduce leakage energy consumption in the proposed cache by turning off the lines that don't have valid data dynamically.

There have been several studies in partitioning the iL1 to several sub-caches. One study is that on the partitioned instruction cache architecture proposed by Kim et al. [9]. They split the iL1 into several sub-caches to reduce per-access energy cost. Each sub-cache in their scheme may contain multiple pages. In contrast, each sub-cache in our scheme is dedicated to only one page, leading to more reduction of per-access energy compared to their work by eliminating tag lookup and comparison in the iL1. They can slightly reduce cache miss rates in some applications, whereas very complex dynamic remapping is required, which may incur negative slack in the critical path. Our scheme can be implemented with much more simple hardware compared to their scheme, which makes it more efficient for embedded processors. Moreover, we also propose the technique to reduce leakage energy consumption in the partitioned cache. A cache design to reduce tag area cost by partitioning the cache has been proposed by Chang et al. [10]. They divide the cache into a set of partitions, and each partition is dedicated to a small number of pages in the TLB to reduce tag area cost in the iL1. However, they did not consider energy consumption. When a request comes into the cache, all partitions are concurrently accessed in their work.

The rest of this paper is organized as follows. Section 2 describes the proposed cache architecture and shows the techniques to reduce dynamic and leakage

energy consumption. Section 3 discusses our evaluation methodology and shows detailed evaluation results. Section 4 concludes this paper.

2 Proposed Power-Aware Instruction Cache

2.1 Reducing Dynamic Energy Consumption

Instruction cache structure focused in this paper is Virtually-Indexed, Physically -Tagged (VI-PT) cache with two-level TLB. VI-PT cache is used in many current processors to remove the TLB access from critical path. In the VI-PT cache, virtual address is used to index the iL1 and the TLB is concurrently looked up to obtain physical address. After that, the tag from the physical address is compared with the corresponding tag bits from each block to find the block actually requested. The two-level TLB, a very common technique in embedded processors (e.g. ARM11 [11]), consists of micro TLB and main TLB. Micro TLB is placed over the main TLB for filtering accesses to the main TLB for low power consumption. When a miss occurs in the micro TLB, additional cycle is required to access the main TLB. When an instruction fetch request from the processor comes into the iL1, virtual address is used to determine the set. If the selected blocks are not valid, a cache miss occurs. If there is a valid block, the tag of the block is compared with the address obtained from the TLB to check whether it was really requested. If they match, the cache access is a hit.

Dynamic energy consumption in the cache is mainly dependent on the cache configuration such as cache size and associativity. In general, small cache consumes less dynamic energy than large cache. However, small cache increases cache miss rates, resulting in performance degradation. Thus, large cache is inevitable for performance. The proposed cache, which we call Power-aware Instruction Cache (PI-Cache), is composed of several small sub-caches in order to make use of both advantages from small cache and large cache. When a request from the processor comes into the PI-Cache, only one sub-cache that is predicted to have the requested data, is accessed. In the PI-Cache, each sub-cache is dedicated to only one page allocated in the micro TLB. The number of sub-caches in the PI-Cache is equal to the number of entries in the micro TLB. Therefore, there is one-to-one correspondence between sub-caches and micro TLB entries.

Increasing the associativity of the cache to improve the hit rates has negative effects on the energy efficiency, because set-associative caches consume more dynamic energy than direct-mapped caches by reading data from all the lines that have same index. In the PI-Cache, each sub-cache is configured as direct-mapped cache to improve the energy efficiency. Each sub-cache size is equal to page size. Therefore, we can eliminate tag array in each sub-cache because all the blocks within one page are mapped to only one sub-cache.

Fig. 1 depicts the proposed PI-Cache architecture. There are three major changes compared with the traditional cache architecture. 1) Id field is added to each micro TLB entry to denote the sub-cache which corresponds to each micro TLB entry. The id field in each micro TLB indicates the sub-cache which all cache blocks within the page are mapped to. 2) There is a register called PSC

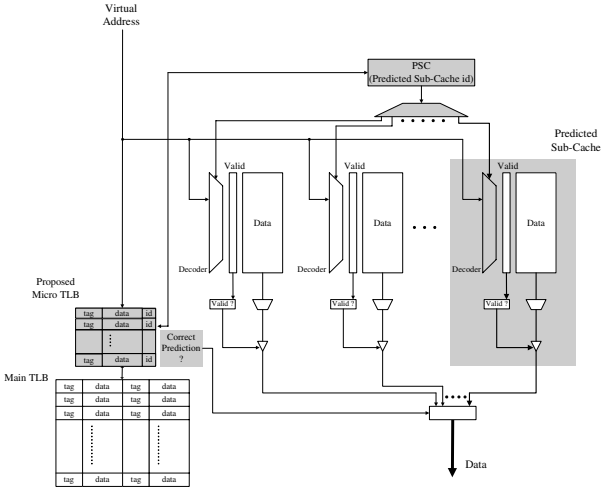


Fig. 1. Proposed Power-aware Instruction Cache Architecture

(Predicted Sub-Cache id) that stores the id of the latest accessed sub-cache. An access to the PI-Cache is performed based on the information in the PSC register. 3) Tag arrays are eliminated in the iL1.

When an instruction fetch request from the processor comes into the PI-Cache, only one sub-cache, which was accessed just before, is accessed based on the information stored in the PSC register. At the same time, the access to the instruction TLB is performed. If the access to the micro TLB is a hit, it means that the requested data is within the pages mapped to the iL1.

In case of a hit in the micro TLB, the id of the matched micro TLB entry is compared with the value in the PSC register to verify the prediction. When the prediction is correct (the sub-cache id corresponding to the matched micro TLB entry is same to that stored in the PSC register), a normal cache hit occurs if data was found in the predicted sub-cache and a normal cache miss occurs if data was not found. A normal cache hit and a normal cache miss mean a cache hit and a cache miss without penalty (another sub-cache access delay), respectively. If the prediction is not correct, it means that the requested data belongs to the other pages in the iL1. In this case, the correct sub-cache is also accessed. This incurs additional cache access penalty. If data is found in the correct sub-cache, a cache hit with penalty occurs. If cache miss occurs even in the correct sub-cache, a cache miss with penalty occurs.

If a miss occurs in the micro TLB, it implies that the requested data is not within the pages mapped to the iL1, consequently a cache miss occurs. In this case, the sub-cache that corresponds to the replaced page from the micro TLB is flushed in whole. Each sub-cache can be easily flushed by resetting valid bits of all cache blocks because the iL1 only allows read operation (No write-back is required). Then, incoming cache blocks which correspond to the newly allocated entry in the micro TLB are placed into the flushed sub-cache.

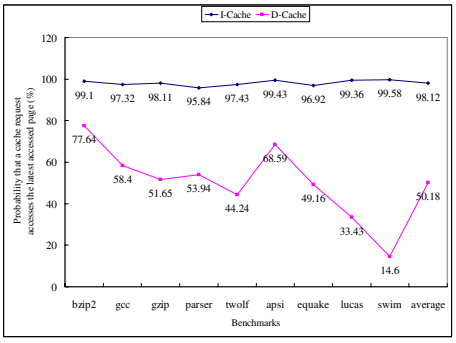


Fig. 2. Probability that a Cache Request Accesses the Latest Accessed Page

The proposed PI-Cache incurs little hardware overhead compared to the traditional cache. Traditional micro TLB must be extended to incorporate the id for each entry for this scheme. However, the number of bits for id field is typically small: 2, 3, or 4 bits. One register is required for the PSC register and one comparator is required to check sub-cache prediction. This overhead is negligible.

The PI-Cache is expected to reduce dynamic energy consumption by reducing the size of accessed cache and eliminating tag comparison. If the hit rates in the PI-Cache do not decrease so much compared to those in the traditional cache, the PI-Cache can be an energy-efficient alternative as an iL1. We expect that the hit rates in the PI-Cache do not decrease so much, based on the following observation. Fig. 2 shows the probability that a cache request accesses the latest accessed page, obtained from our simulations using SimpleScalar [12]. As shown in the graph, almost all cache requests coming into the iL1 are within the latest accessed page. This feature can be attributed to high temporal/spatial locality of applications. The proposed PI-Cache is not applicable to data cache, because the locality in data cache is inferior to that in instruction cache, as shown in Fig. 2.

2.2 Reducing Leakage Energy Consumption

Traditionally, the major component of energy in current technology is dynamic energy consumption. However, leakage energy is expected to be more significant as threshold voltages decrease in conjunction with smaller supply voltages in upcoming chip generations [13].

In the proposed PI-Cache, there is no conflict miss, since one page is mapped to one sub-cache whose size is equal to page size. However, there are more compulsory misses than the traditional cache, since the sub-cache in the PI-Cache is flushed whenever the corresponding entry in the micro TLB is replaced. Compulsory misses may be eliminated if the PI-Cache transfers all cache blocks in the page simultaneously when the page is allocated in the micro TLB. However, the PI-Cache does not transfer whole page at the same time, because it may incur serious bus contention and energy problem. The PI-Cache transfers the

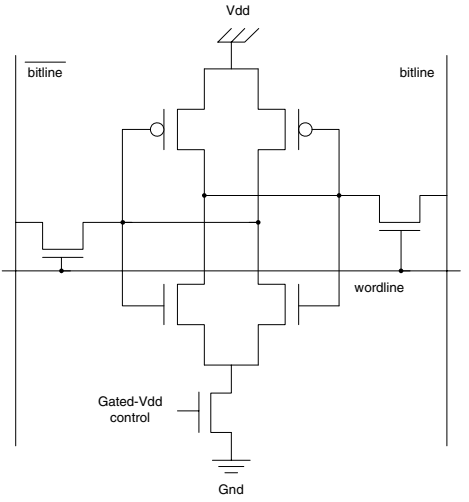


Fig. 3. SRAM cell with a NMOS gated-Vdd

cache block from lower level memory only when it is requested. Therefore, there may be many lines that don't have valid data in the PI-Cache. Based on this assumption, we present the technique to reduce leakage energy consumption in the PI-Cache by turning off the unused memory cells in the PI-Cache.

To turn off the lines in the PI-Cache, we use the circuit level technique called gated-Vdd proposed by Powell et al. [5], shown in Fig. 3. Key idea in this technique is to insert a 'sleep' transistor between the ground and the SRAM cells of the cache line. When the line is turned off by using this technique, leakage energy dissipation of the cache line can be considered negligible with a little area overhead, which is reported to be about 5% [5].

In the PI-Cache, if a miss occurs in the micro TLB, the sub-cache that corresponds to the replaced page from the micro TLB must be flushed. At this time, we turn off all cache lines in the flushed sub-cache by using gated-Vdd technique. At the time of initialization, all cache lines in the PI-Cache are turned off to save leakage energy consumption. Then, each cache line will be turned on when a first access to the line is requested. In other words, each cache line is turned on during reading data from lower level memory after compulsory miss. After that, each cache line will not be turned off until the sub-cache is flushed. At the time of first access to each cache line, a compulsory miss occurs inevitably. Therefore, leakage energy consumption in the PI-Cache is expected to be reduced with no performance degradation by using this technique. Moreover, no more hardware overhead except the gated-Vdd is required for this technique.

3 Experiments

In this section, we show the simulation results to determine the characteristics of the proposed PI-Cache with respect to the traditional instruction cache.

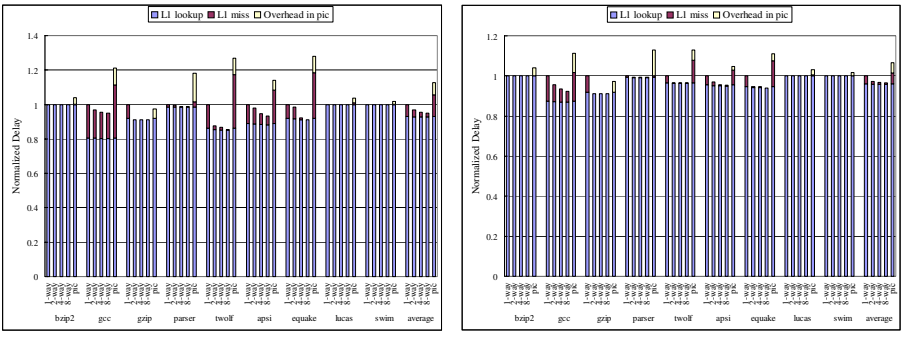
Table 1. Memory Hierarchy Parameters

Parameter	Value
Micro TLB	fully associative, 1 cycle latency
Main TLB	32 entries, fully associative, 1 cycle latency, 30 cycle miss penalty
L1 I-Cache	16KB and 32KB, 1-way – 8-way, 32 byte lines, 1 cycle latency
Sub-cache in the PI-Cache	4KB (Page size), 1-way, 32 byte lines, 1 cycle latency
L1 D-Cache	32KB, 4-way, 32 byte lines, 1 cycle latency, write-back
L2 Cache	256KB unified, 4-way, 64 byte lines, 8 cycle latency, write-back
Memory	64 cycle latency

Simulations were performed on a modified version of SimpleScalar toolset [12]. The power parameters were obtained from CACTI where we assumed 0.18um technology [14]. The simulated processor is a 2-way superscalar processor with an L2 cache, which is expected to be similar to the next generation embedded processor by ARM [15]. Simulated applications are selected from SPEC CPU2000 suite [16]. Memory hierarchy parameters used in the simulation are shown in Table 1.

3.1 Performance

Fig. 4 shows the normalized instruction fetching delay obtained from simulations. We assume that *pic* in the graphs denotes the proposed PI-Cache. *L1 lookup* portion in the bar represents the cycles required for iL1 accesses. *L1 miss* portion denotes the cycles incurred by iL1 misses. *Overhead in pic* portion denotes the delayed cycles incurred by sub-cache misprediction in the PI-Cache scheme, namely the penalty to access another sub-cache after misprediction. Note that traditional cache schemes do not have *Overhead in pic* portion.



(a) 16KB L1 I-Cache (b) 32KB L1 I-Cache

Fig. 4. Normalized Cache Delay

Table 2. Physical Cache Access Time

Access Time, ns	1-way	2-way	4-way	8-way	PI-Cache
16KB	1.129	1.312	1.316	1.383	0.987
32KB	1.312	1.455	1.430	1.457	0.987

As shown in these graphs, set-associative caches show less cache delay than direct-mapped caches by reducing the delay due to cache misses. Therefore, the cache delay is reduced with more degree of associativity. As shown in Fig. 4, the performance of 16KB PI-Cache is degraded by 12% on the average compared to that of the traditional direct-mapped cache. 32KB PI-Cache is degraded by 6% on the average. This performance degradation is caused by two reasons: one is the degradation of the hit rates by restricting the blocks to be allocated in the iL1 to the blocks within the pages mapped to the micro TLB entries. The other is the sub-cache misprediction which incurs additional sub-cache access delay, indicated by the *Overhead in pic* portion. Performance gap between the traditional caches and the PI-Cache decreases as the cache size increases. This is because the hit rates in the PI-Cache improve by increasing the number of the pages mapped to the iL1: 32KB PI-Cache with 8 pages (sub-caches) compared with 16KB PI-Cache with 4 pages (sub-caches). From these results, the PI-Cache is expected to be more efficient as the cache size becomes larger.

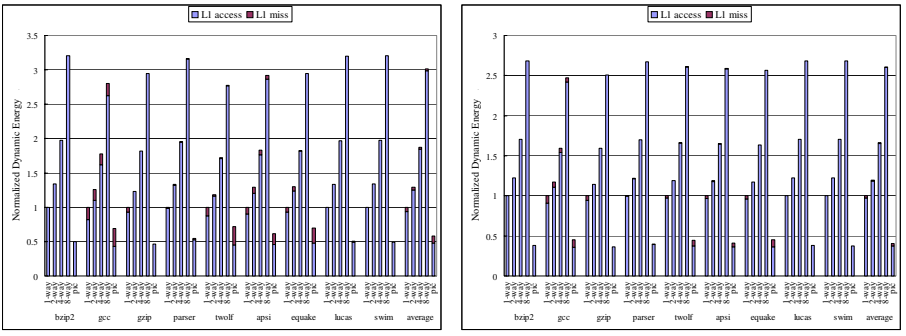
The results shown in Fig. 4 are obtained from the configurations in Table 1. We simulated all cache configurations with same cache access latency (1 cycle). In fact, physical access time varies by the cache configurations. Table 2 gives the physical access time according to each cache models obtained from CACTI model. Physical access time of the PI-Cache is "1 AND gate delay (it is required to enable the sub-cache indicated by the PSC register, 0.114 ns, obtained from ASIC STD130 DATABOOK by Samsung Electronics [17]) + Sub-cache access latency (0.873 ns, obtained from CACTI)". As shown in Table 2, physical access time of the PI-Cache is faster than that of the direct-mapped traditional cache, because accessed cache size is small and tag comparison is eliminated in the PI-Cache. This feature is well shown in 32KB iL1 than 16KB iL1. Therefore, if the processor clock speeds up or the size of cache increases in the future, the proposed PI-Cache is expected to be more favorable.

3.2 Dynamic Energy Consumption

Table 3 shows per-access energy consumption according to each cache models obtained from CACTI model. In the traditional caches, the energy consumed

Table 3. Per-access Energy Consumption

Energy, nJ	1-way	2-way	4-way	8-way	PI-Cache
16KB	0.473	0.634	0.935	1.516	0.232
32KB	0.621	0.759	1.059	1.666	0.232



(a) 16KB L1 I-Cache (b) 32KB L1 I-Cache

Fig. 5. Normalized Dynamic Energy Consumption

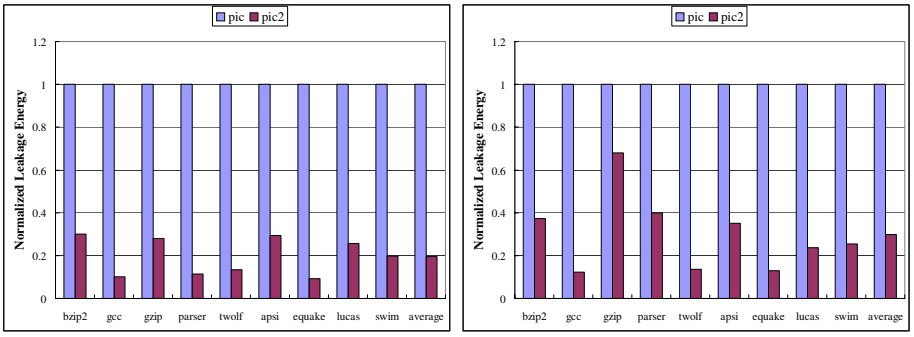
by the cache increases as the degree of associativity increases. The increase in associativity implies the increase of output drivers, comparators, sense amplifiers, consequently the increase of total energy. The PI-Cache consumes less per-access energy compared to traditional caches. There are two reasons for better energy efficiency: one is that the size of cache accessed in the PI-Cache is smaller than the traditional cache by partitioning it to several sub-caches. The other is the elimination of accesses to tag arrays in the PI-Cache.

Detailed energy consumption obtained from SimpleScalar and CACTI together is shown in Fig. 5. Total energy consumption presented in these graphs is the sum of the dynamic energy consumed during instruction fetching. In Fig. 5, *L*1 *access* portion denotes the dynamic energy consumed during iL1 accesses, and *L*1 *miss* portion represents the dynamic energy consumed during accessing lower level memory incurred by misses in the iL1. 16KB PI-Cache gives the improvement of energy efficiency by 42% on the average and 32KB PI-Cache reduces energy consumption by 59% on the average. As expected, the PI-Cache is more energy efficient with a large cache.

3.3 Leakage Energy Consumption

Fig. 6 shows the normalized leakage energy consumption in the iL1. In these graphs, *pic* denotes the original PI-Cache scheme and *pic*2 denotes the PI-Cache scheme adopting the technique to reduce leakage energy consumption. Each bar in the graphs is obtained by assuming that leakage energy is proportional to ((Average number of lines turned on * Average turned on time)/(Total number of lines in the iL1 * Execution time)), based on the results from simulations with SimpleScalar.

According to these graphs, the proposed technique reduces leakage energy consumption in the 16KB PI-Cache by 80% on the average. It reduces leakage energy consumption in the 32KB PI-Cache by 70% on the average. Leakage energy reduction is well shown in 16KB PI-Cache. This is because each sub-cache in 16KB PI-Cache is flushed more frequently compared to that in 32KB PI-Cache.



(a) 16KB L1 I-Cache (b) 32KB L1 I-Cache

Fig. 6. Normalized Leakage Energy Consumption

From these observations, we can realize that much portion of the PI-Cache is turned off during execution. The sub-cache size in the PI-Cache can be adjusted to be smaller than the page size to reduce these unused memory cells. However, if we decrease the sub-cache size, there must be tag comparison in the PI-Cache and the control logics have to be inserted. Moreover, performance degradation will be incurred by conflict misses in the sub-cache. The method to decrease the sub-cache size with little effects on the performance and the energy efficiency could be a promising direction for future work.

4 Conclusions

We have described and evaluated a Power-aware Instruction Cache (PI-Cache) architecture, which splits the L1 instruction cache into several small sub-caches for reducing energy consumption in a processor. When a request from the processor comes into the PI-Cache, only one predicted sub-cache is accessed and tag comparison is eliminated, which leads to dynamic energy reduction. We also proposed the technique to reduce leakage energy consumption in the PI-Cache, which turns off the lines that don't have valid data dynamically. The proposed technique reduces leakage energy consumption with no performance degradation. According to the simulation results, the proposed techniques reduce dynamic energy by 42% – 59% and reduce leakage energy by 70% – 80%. Therefore, the PI-Cache is expected to be a scalable solution for a large instruction cache in embedded processors.

References

1. Segars, S.: Low power design techniques for microprocessors. In: Proceedings of International Solid-State Circuits Conference. (2001)
2. Kin, J., Gupta, M., Mangione-Smith, W.: The filter cache: An energy efficient memory structure. In: Proceedings of International Symposium on Microarchitecture. (1997) 184–193

3. Bellas, N., Hajj, I., Polychronopoulos, C.: Using dynamic cache management techniques to reduce energy in a high-performance processor. In: Proceedings of Low Power Electronics and Design. (1999) 64–69
4. Albonesi, D.H.: Selective cache ways: On-demand cache resource allocation. In: Proceedings of International Symposium on Microarchitecture. (1999) 70–75
5. Powell, M., Yang, S.H., Falsafi, B., Roy, K., Vijaykumar, T.N.: Gated-vdd: A circuit technique to reduce leakage in deep-submicron cache memories. In: Proceedings of Low Power Electronics and Design. (2000) 90–95
6. Kaxiras, S., Hu, Z., Martonosi, M.: Cache decay: Exploiting generational behavior to reduce leakage power. In: Proceedings of International Symposium on Computer Architecture. (2001) 240–251
7. Flautner, K., Kim, N.S., Martin, S., Blaauw, D., Mudge, T.: Drowsy caches: Simple techniques for reducing leakage power. In: Proceedings of International Symposium on Computer Architecture. (2002) 148–157
8. Kim, N.S., Flautner, K., Blaauw, D., Mudge, T.: Drowsy instruction cache: Leakage power reduction using dynamic voltage scaling and cache sub-bank prediction. In: Proceedings of International Symposium on Microarchitecture. (2002) 219–230
9. Kim, S.T., Vijaykrishnan, N., Kandemir, M., Sivasubramaniam, A., Irwin, M.J.: Partitioned instruction cache architecture for energy efficiency. ACM Transactions on Embedded Computing Systems **2** (2003) 163–185
10. Chang, Y.J., Lai, F., Ruan, S.J.: Cache design for eliminating the address translation bottleneck and reducing the tag area cost. In: Proceedings of International Conference on Computer Design. (2002) 334
11. ARM: (ARM1136J(F)-S) http://www.arm.com/products/CPUs/ARM1136JF-S.html.
12. Burger, D., Austin, T.M., Bennett, S.: Evaluating future microprocessors: the simplescalar tool set. Technical Report TR-1308, Univ. of Wisconsin-Madison Computer Sciences Dept. (1997)
13. Borkar, S.: Design challenges of technology scaling. IEEE Micro (1999) 23–29
14. Shivakumar, P., Jouppi, N.P.: Cacti 3.0: An integrated cache timing, power, and area model. Technical Report TR-WRL-2001-2, Univ. of Wisconsin-Madison Computer Sciences Dept. (2001)
15. Muller, M.: At the heart of innovation. (http://www.arm.com/miscPDFs/6871.pdf)
16. SPEC: (SPEC CPU2000 Benchmarks) http://www.specbench.org.
17. SamsungElectronics: Asic std130 databook. (http://www.samsung.com)

Dynamic Voltage Scaling for Power Aware Fast Fourier Transform (FFT) Processor

David Fitrio, Jugdutt (Jack) Singh, and Aleksandar (Alex) Stojcevski

Centre of Telecommunications and Microelectronics,
Victoria University of Technology, P.O. Box 14428, Victoria 8001, Australia
david.fitrio@research.vu.edu.au

Abstract. In recent years, power dissipation in CMOS circuits has grown exponentially due to the fast technology scaling and the increase in complexity. Supply Voltage scaling is an effective technique to reduce dynamic power dissipation due to the non-linear relationship between dynamic power and Vdd. In other words, Vdd can be scaled freely except with limitation for below threshold voltage operation. The dynamic voltage scaling architecture mainly consists of dc-dc power regulator which is customised to produce variability on the Vdd. The implemented architecture can dynamically vary the Vdd from 300 mV to 1.2V, with initial setup time of 1.5 µsec. This paper investigates the effect of DVS on dynamic power dissipation in a Fast Fourier Transform multiplier core. Implementation of DVS on the multiplier blocks has shown 25% of average power reduction. The design was implemented using 0.12µm ST-Microelectronic 6-metal layer CMOS dual- process technology.

1 Introduction

Reduction of power dissipation in CMOS circuits requires addressing in numerous areas. Starting from the selection of the appropriate transistor library to minimise leakage current, implementation of low power design architectures, power management implementation, and chip packaging are very critical processes in developing power efficient devices. Another critical consideration of low power silicon design is energy/power-aware system. A Power-aware system should be able to decide its minimum power requirements by dynamically scaling its operating frequency, supply voltage and/or threshold voltage according to different types of operating scenarios.

Typically, System on a Chip (SoC) implementation of a device utilises general purpose hardware, such as FPGA, DSP or embedded systems. General purpose hardware has allows faster time to market, with ease of programmability and interface. However, these hardware devices can be power inefficient. The increasing product demand for application specific integrated circuit or processor for independent portable devices has influenced designers to implement dedicated processors with ultra low power requirements. One of the dedicated processors is Fast Fourier Transform (FFT), which is vastly used in signal processing.

FFT has always been a popular algorithm due to its efficiency in computing Discrete Fourier Transform (DFT) of time series [1]. In the early days, implementing an FFT algorithm on a chip was limited due to large hardware area, power consumption and low speed signal. FFT processors have been commonly used in signal analysis

T. Srikanthan et al. (Eds.): ACSAC 2005, LNCS 3740, pp. 52 – 64, 2005.

and processing in numerous applications such as, image and voice/speech processing, telecommunication and biomedical applications. In emerging areas such as wireless sensor telecommunication network or in portable biomedical fields, the demand of portable devices with extended battery life is extremely high. A wireless network, for example, requires up to thousands of sensor nodes, which should be able to maintain their operating life with only ambient power or should last for a few years with battery power. Therefore, the main concern of these applications is the ultra low power dissipation rather than high computation speed.

2 CMOS Power Dissipation

As CMOS transistors scale down to provide smaller dimension and better performance, scaling down transistor sizes has achieved an additional 60% increase in frequency and reduction in delay. However, the complexity of the system has also pushed up the total power dissipation.

The total power dissipation in CMOS circuits is the accumulation of dynamic and static power dissipation, as represented by Equation 1 [2].

$$P_{total} = P_{dynamic} + P_{static} \tag{1}$$

Due to its nature, dynamic power dissipation ($P_{dynamic}$) is more dominant in majority of applications. $P_{dynamic}$, has a direct relationship between switching frequency and supply voltage (V_{dd}),

$$P_{dynamic} = f_{0\to1}.f_{clk}.V_{dd}{}^{2}.C_{L} \tag{2}$$

where, $f_{0\to1}$ is the switching frequency, f_{clk} is the clock frequency and C_L is the capacitance on a node.

However, static power dissipation (P_{static}) has a direct relationship to the subthreshold leakage current, which increase exponentially with decreasing Vt for given Vgs [3]. The exponential increase of subthreshold leakage drain current with decreasing V_t for a given V_{gs} is shown in Equation 3. In other words, a transistor with higher threshold voltage has a lower leakage current. Generally, the static power dissipation can be expressed as:

$$P_{static} = V_{dd}.I_{subthreshold}.e^{\frac{(V_{gs}-V_t)}{nV_{TH}}} \tag{3}$$

where n is the process parameter, V_t is the threshold voltage and V_{TH} is the thermal voltage at room temperature.

Theoretical and experimental procedures show that lowering V_{dd} directly reduces the power consumption. However, the drawback in lowering V_{dd} is a longer delay, which signifies slower performance or processing speed in integrated circuits [4-7]. This is undesirable in high performance and high speed systems.

One of the proposed solutions to lower total power dissipation is to use dynamic voltage (V_{dd}) scaling (DVS). Dynamic voltage scaling enables power supply scaling into various voltage levels instead of a fixed supply voltage. The main concept is that

if the speed of the main logic core can be adjusted according to input load or amount of processor's computation required, then the processor can run into "just enough" operation to meet the throughput requirement, as shown in Fig. 1.

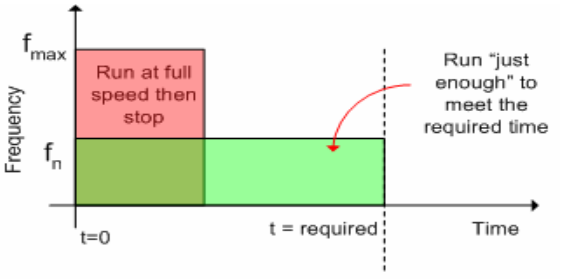


Fig. 1. "Just enough" operation in dynamic V_{dd} scaling

2.1 Power Management with Dynamic Voltage Scaling (DVS)

The facts behind power reduction in DVS are:

- A system is not always required to work 100% performance.
- The total power dissipation is dominated by dynamic power.

The power reduction concept of DVS is similar to that of variable threshold scaling (V_{ts}). Instead of adjusting the threshold voltage, DVS scheme adjusts V_{dd} according to throughput required. The supply voltage can be reduced for processes, such as background task, which can be executed at a reduced frequency thus minimizing power consumption. The performance level is reduced during low utilization periods in such way that the processor finishes its task "just in time" by reducing the working frequency. While the operational frequency is lowered, at the same time the supply voltage V_{dd}, could also be reduced.

Current DVS implementation is focused more at software level which resides in the kernel of the operating system rather than hardware implementation. The portability of DVS system to support multiple platforms with different requirements is the main concern with many DVS designer. These considerations make DVS designers to prefer on DVS implementation at software level. However, optimisation in DVS architecture can still be done by realising some of the modules in hardware to reduce the power consumption of the DVS system.

3 DVS System Architecture

This section of the paper discusses the system modules required for the power aware FFT processor with DVS. The power management module discussed is the DVS architecture with its building blocks which include DAC, tunable ring oscillator, pulse width modulator, phase frequency detector, current driver and loop filter are employed.

Generally, there are three key components for implementing DVS in a system with central processing unit (CPU):

- An operating system capable in varying the processor speed.
- A regulator which generates a minimum voltage for a particular frequency.
- A central processing unit which is capable of working on wide operating voltage.

As described above, hardware implementation alone would not be possible. The processor speed is controlled by the operating system, where an obligation is to gather the load requirements of the process from a power profiler module. In order to minimise energy dissipation, the supply voltage must vary as the processor frequency varies. However, the operating system is not capable of controlling the required minimum supply voltage for a given frequency. A hardware implementation is required to provide this functionality. The overall architecture of a DVS system is presented in Fig. 2.

The digital-to-analog (DAC) converter in the DVS system converts the quantised digital bits from the CPU's load to an analog voltage. The represented analog voltage is then used by the Pulse Width Modulator (PWM) as voltage reference together with reference external clock pulses to generate a series of train pulses. The pulses vary in period, depending upon the voltage level from DAC.

Phase Frequency Detector (PFD) performs the comparison between the incoming PWM signal and the ring oscillator output which will then trigger the charge pump to give different level of V_{dd}. However, a clean and stable voltage supply level is required in most application for reliable performance. A loop filter filters out the ripples from the charge pump to produce a stable supply voltage for the main system.

The DAC used in this system is a weighted transistor voltage biased current DAC. The transistor network configuration used in this DAC is based on R-2R architecture. By replacing the resistors with transistors of W/L and 2W/L, the same R-2R characteristic can be obtained. The implementation of a transistor-only DAC will reduce the power dissipation of the DVS system. The input bits (Bit<n:0>) controls the charging and discharging of the weighted transistor.

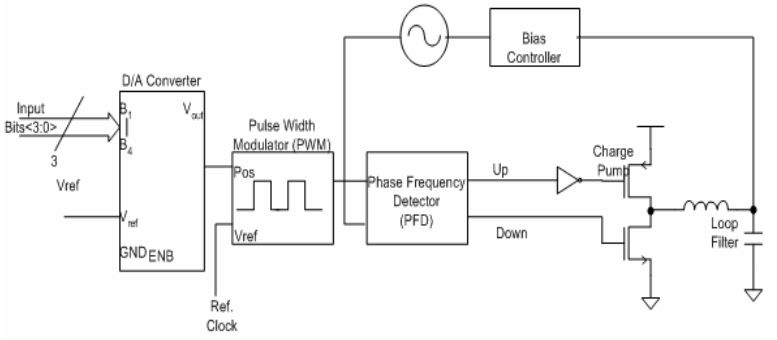


Fig. 2. DVS Implementation System Architecture

Pulse width modulation is a common modulation technique in signal processing. PWM generates a series of pulses where the pulse width varies depending upon the weighted input binary scalar. The PWM module used in this DVS system has the same functionality as the conventional system. The module generates a series of

weighted pulses from the comparison of Vref, as the reference voltage, and the oscillation frequency from the clock reference. The series of pulses from PWM is then compared by Phase and Frequency Detector (PFD) with tunable oscillator frequency.

PFD is commonly used as phase detector in frequency lock in Phase Locked Loop (PLL) systems. As the phase difference of the two input signals, Ref-Clk (coming from PWM output) and Clk (the actual VCO clock), change so do the two output signals: Up and Down. The bigger the phase difference, the larger is the pulse width produced at the Up and Down terminals. The output signal DOWN is high when Clk leads the Ref-Clk signal, and output signal UP is high when Clk lags the Ref-Clk. These series of small pulses control the charge current injected by the charge pump circuit.

The purpose of charge pump circuit is to transform time domain train pulses into continuous steady voltage for VCO, depending on the signals from PFD. If reference clock signal lags the VCO signal, PFD will discharge the charge pump and lower the output voltage, and vice versa. The loop filter removes jitters and smoothes out the continuous steady voltage from the charge pump into analog voltage for VCO frequency control.

The VCO used in this DVS architecture is a ring oscillator which consists of transmission gates switches. A wide frequency range can be generated by operating one of the switches on. The minimum oscillation frequency is obtained by using all the inverter stages. The ring oscillator also comprises of a current controlled inverter connected in parallel with conventional inverter for gain control and for different inverter frequency stages. The output frequency of the VCO can be programmed from 100 KHz to 334 MHz.

4 Fast Fourier Transform

FFT is an effective algorithm in processing Discrete Fourier Transform (DFT). A general N-point DFT is defined by Equation (5) as [8],

$$X(k) = \sum_{n=0}^{N-1} x(n) W_N^{nk},\ \text{k=0...N-1} \tag{4}$$

where $x(n)$ is the DFT sequence and both $x(n)$ and $X(k)$ are complex numbers. The twiddle factor W_N^{nk}, represented as

$$W_N^{nk} = e^{-j\left(\frac{2\pi nk}{N}\right)} \approx \cos\left(\frac{2\pi nk}{N}\right) - j\sin\left(\frac{2\pi nk}{N}\right) \tag{5}$$

is used to divide the DFT into successively smaller DFTs by using the periodical characteristics of complex exponential. The twiddle factor is used in FFT to increase the computational efficiency. Therefore, the decomposition of splitting the DFT input sequence $x(n)$ into smaller subsequence is called decimation in time (DIT). Inversely, decomposing the output sequence $X(k)$ is called decimation in frequency (DIF).

The basic calculations in FFT are the multiplication of two complex input data with the FFT coefficient W_N^{nk}, at each datapath stage, followed by their summation or subtraction with the associated data, as shown in Fig. 3 [8].

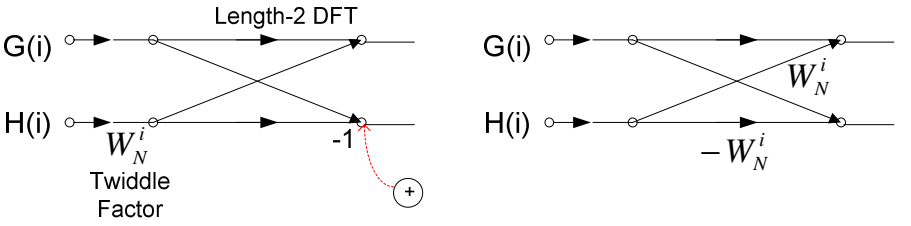


Fig. 3. Radix 3 DIT Equivalent Butterfly Unit Operation

The calculation complexity is specified by $O(N^2)$ number of multiplication and $N(N-1)$ number of addition required. The number of computational complexity can be reduced to $O(N \log N)$, if the FFT algorithm is applied, as shown in Fig. 4.

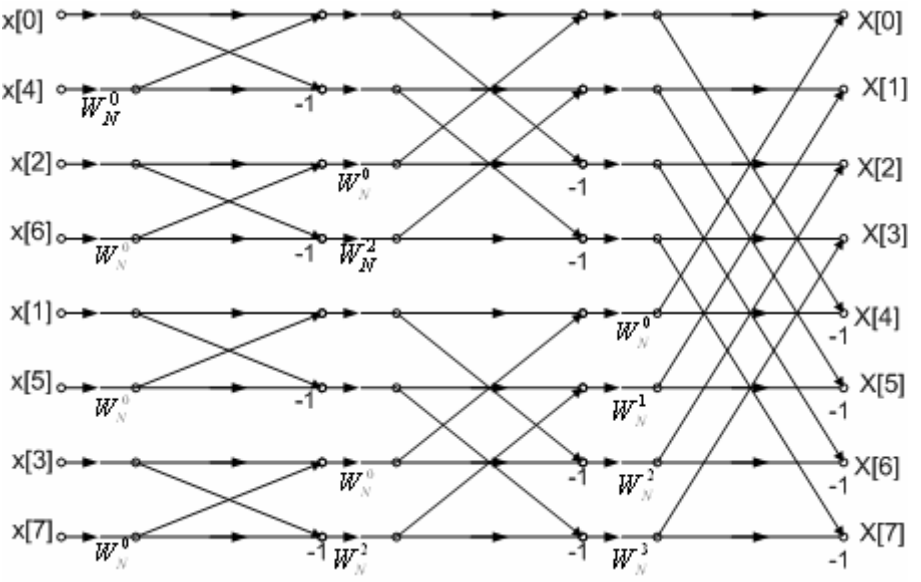


Fig. 4. Point DIT FFT data flow

The number of sampling points required is specified by the power of the radix-r in the FFT algorithm. Although, radix-2 algorithm could cover most of the possible sampling points required, radix-2 algorithm requires more computation cycles than, for example, radix-4. Contrarily, radix-4 could not handle other sampling points

which are not powers of four, such as 128, 512, etc. Therefore, a mixed radix (MR) algorithm that uses both radix-2 and radix-4 to solve FFT which are not powers of four could be used [9].

Generally, N-point of FFT computation requires $(N/r)\times\log_r N$ radix-r butterfly computation. Regardless, the computation could be done by a butterfly unit on each stage (pipelined) or a single recursive butterfly unit (shared memory) depending on the application requirements.

4.1 FFT Processor Architectures

Numerous FFT processor architectures have been developed based on Cooley-Turkey algorithm for different applications. Pipeline architecture, for example, is widely used in high throughput applications [10]. However, the disadvantage of this architecture is the relatively large silicon area, due to log_rN process elements, where N represents the FFT length and r represents its radix, which are required by pipelined architecture.

Another different FFT architecture is the shared memory architectures [11]. The advantages of shared memory architectures are area-efficient and lower overall power consumption. However, the shared memory architectures could not achieve a high speed operation, due to more computation cycles required. The main trade-offs in the FFT processor is hardware overhead and speed requirements.

The low power and low speed characteristics of the shared memory FFT processor architecture make it suitable for a low power application such as in biomedical applications. FFT algorithm is commonly used in biomedical field such as in data acquisition of biomedical signals [12] , ultrasound image processing[13], heart sound signal monitoring[14] and hearing aid applications[15].

Shared memory FFT processor architecture generally consists of a Butterfly core datapath, a data storage memory and a twiddle factor look up table, as shown in Fig. 5.

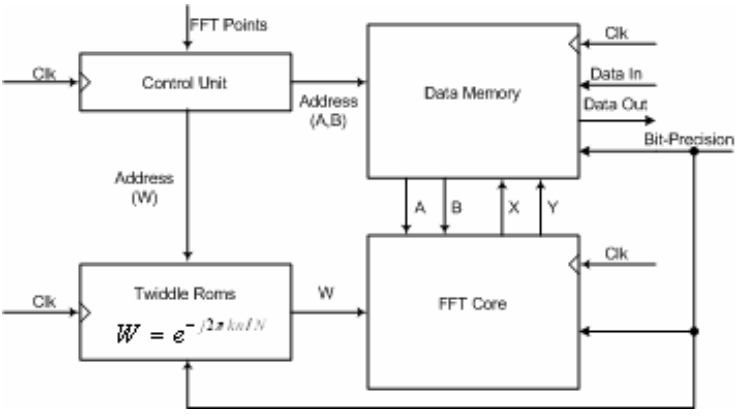


Fig. 5. A Dedicated FFT Processor Architecture

The main computational core is handled by a Butterfly core radix. Conventionally, the computational core in a dedicated FFT processor can range from a single multiplier algorithm logical unit to a high order radix FFT.

In this paper, the FFT core is simulated with the DVS system comprise of 8 bits Baugh Wooley multiplier. Baugh Wooley Multiplier (BW) [16] is used for two's complement multiplication due to its efficiency in handling signed bits. The effectiveness in handling signed bits multiplication makes it a common processing core in FFT processors. Baugh Wooley algorithm has been selected for the FFT processor presented in this paper, due to:

- simplicity, regularity and modularity of the structure;
- implementation flexibility on desired length decomposition.

The required function in an FFT butterfly is $X=G(i)+H(i)\times W_N^k$ and $Y=G(i)+H(i)\times W_N^k$, where G(i), H(i), and W_N^{nk} are complex inputs, and X and Y are the complex outputs. This is performed for (log2N-1) stages of the N-point real value FFT.

In this paper, an implementation of DVS with a shared memory architecture FFT processor for biomedical application is demonstrated. The key characteristic of small area, ultra low power, low throughput but maintaining low error rate and efficiency is the challenge in FFT processor for biomedical applications. Fig. 6 shows the logic diagram for a 4-bit BW multiplier performing two's complement arithmetic and Fig. 7 shows the gate level implementation of the multiplier.

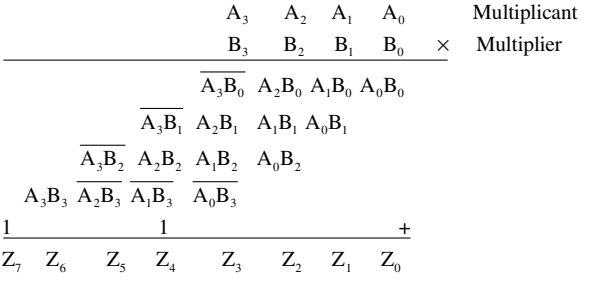


Fig. 6. 4 x 4 bit 2's complement multiplication using the Baugh Wooley Algorithm

Another important component in shared memory FFT processor architecture is the memory cells. Seventy-five percent of the total power consumption in a FFT processor belongs to memory cells data access and the complex number multiplier operation. It is also understandable that the larger the number of bits in the multipliers or in a long size FFT the larger word-length required for the memory cells. Memory cells require huge chip area with large power consumption. Another aspect that determines power consumption in memory cells is the number of access ports. A single port memory access can be efficient in regards to power dissipation, however it also means a bottle neck in high speed FFT operation.

The MTCMOS or Multi-Threshold CMOS topology was chosen for the memory architecture technique in the FFT processor due to the number of inactive (standby)

cells due to the multiplier processing speed. Initially, the MTCMOS principal was applied in the design of SRAM to reduce the power dissipation of the peripheral circuits such as row decoders and input/output buffers.

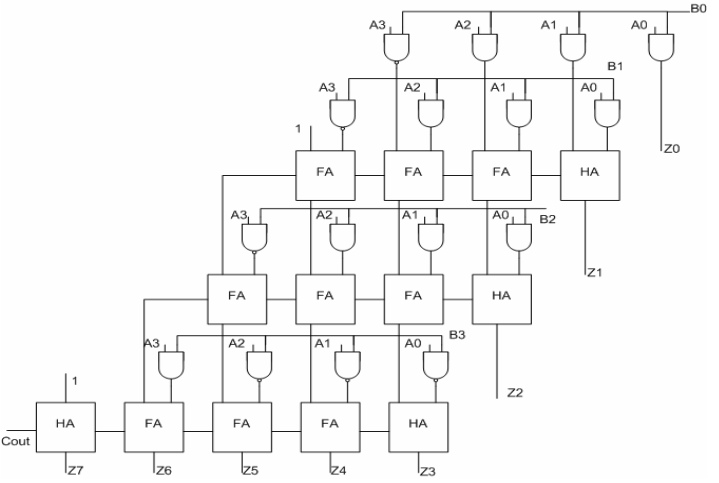


Fig. 7. Gate level implementation of 4x4 bit Baugh Wooley Multiplier Gates Representation

MTCMOS is a circuit technique that uses two different combinations of transistors type. Low- V_t transistors for the high-speed core logic and High-V_t transistors as power switch to reduce leakage current. The main principle of MTCMOS is shown in Fig. 8. MTCMOS has been a popular technique because of simplicity of the design. Ideally the larger the threshold level the lower the leakage current, however, one must decide the optimum value of threshold level between the power switch (High- V_t devices) and (Low- V_t devices), as recovery delay tends to increase in higher threshold level . A power switch with thicker oxide (t_{ox}) must be considered to prevent source-drain current blow up.

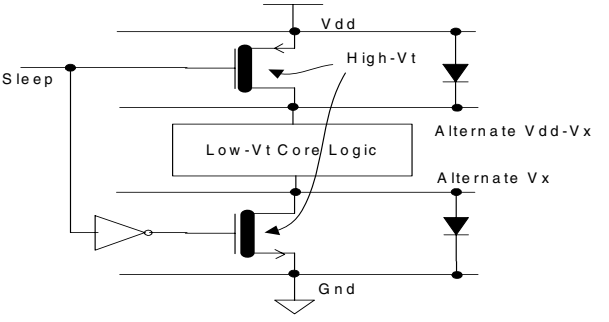


Fig. 8. MTCMOS circuit architecture principal

In this paper, the MTCMOS SRAM was designed by using the conventional gated-V_{dd} and Gnd structure, which was introduced in [17, 18]. This technique reduces the leakage current by using the conventional method of high-V_t transistors between V_{dd} and Gnd to cut off the power supply of the low-V_t memory cell, when the cell is in sleep mode. However, modification was done by applying an addition virtual V_{dd} and Gnd lines for data loss prevention, as shown in Fig. 9a). The two virtual lines will maintain the stored charge of the memory cells while the power lines are cut off. This technique introduces a slight delay in write and read time due to activation of sleep transistors. The delay is necessary for the memory cells to recover from sleep mode to active mode. The read and write operation signals of MTCMOS SRAM is shown in Fig. 9b).

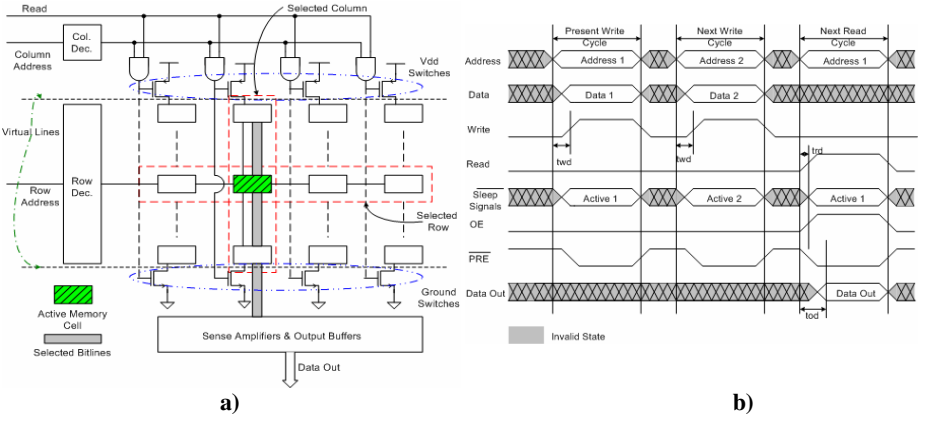


Fig. 9. MTCMOS a) Memory cells array structure, b) Operation signals in read and write cycles

5 Results

The DVS system, multiplier and the SRAM were designed and simulated in Cadence Design Framework II Analog Environment using 0.12 low leakage ST-Microelectronic library. The simulated waveform of the initial voltage startup time is depicted in Fig. 10. It illustrates the 1.5us start-up time of the DVS system, which occurs at an initial voltage of 300mV to a settling voltage of 1.2V. The system startup time varies depending on the size of the transistors in the charge pump, the loop filter, and the load connected to the DVS system.

The voltage scaling occurring at the output, which intern is the supply voltage of the FFT, occurs due to the variation of the VCO frequency. This phenomenon is illustrated in Fig. 11. The variation of the VCO frequency transpires because of the DAC weighted binary input. The external clock shown in Fig. 11a, is set to 150MHz, while the VCO frequency, shown in Fig. 11e, is adjusted to be slightly above 150MHz. It can be clearly seen in Fig. 11b, prior to the VCO frequency reaching 1us, the DVS output voltage (V_{dd}) ramps up to 1.2V, as the full set of binary inputs (i.e. Bit<4:0> are all 1) are

given. In addition to this, the supply voltage output in Fig. 11b, only ramps up to 700mV as the VCO frequency is reduced below the external clock of 150MHz.

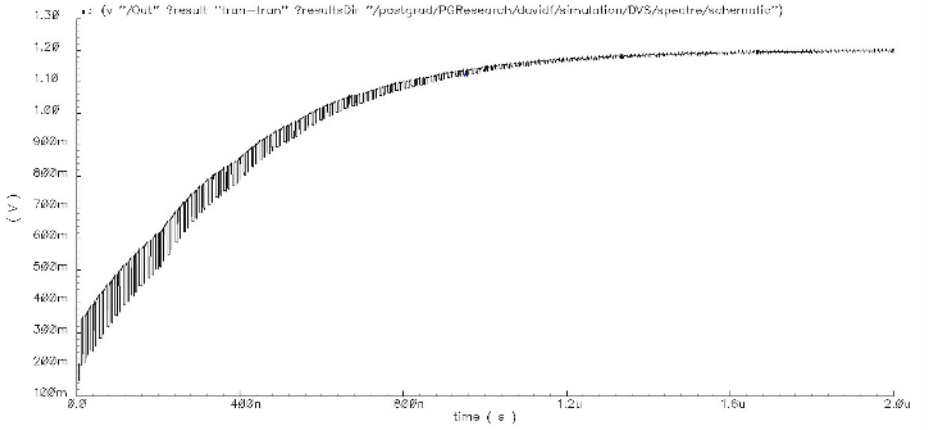


Fig. 10. System Initial Setup Time

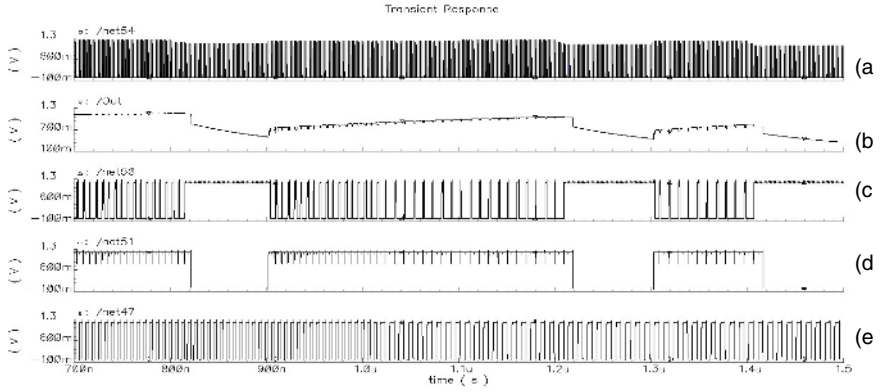


Fig. 11. V_{dd} Scaling with Frequency Variation

The performance of the DVS system is summarised in Table 1. Overall the DVS system without any load consists of 366 transistors and dissipates 174.1uW at 150 MHz operation frequency. DVS implementation on the FFT multiplier core is summarised in Table 2. The simulation results on Table 2 indicate that implementing DVS in a larger multiplier is more effective with average power reduction of approximately 25%.

The performance of 256 bits SRAM with MTCMOS for leakage reduction during standby operation was simulated and is presented in Table 3.

Table 1. DVS Performance Summary

Fabrication Technology	ST-Microelectronic 0.12 um 6-metal layer LLUL library dual-process CMOS
Number of Transistor	366 transistor
Power Supply	1.2 V
Power Dissipation **Scaling Steps**	174.1uW @ 150MHz 4 Bits of VCO frequency and 4 Bits of DAC Binary Input. Total Steps of 256 steps

Table 2. Power reduction comparison with different number of multiplier bits

Multiplier bits (n)	**Power (uW)**		**Reduction in Power (%)**
	Without DVS	With DVS	
8	1.47	1.12	23%
16	3.42	2.55	25%
32	7.81	5.68	27%
64	18.6	12.8	31%

Table 3. MTCMOS SRAM simulation result

SRAM Simulation Results		
Techniques	Conventional	MTCMOS
SRAM Cell Size	8 bits word × 16 bits (256 bits)	
Minimum Cycle Time (@ 1.8 V)	2 ns	2.2 ns
Total Power Dissipation (@ typical 1.2 V, f = 300Mhz)		
Standby Power	[illegible] FΩ	[illegible] FΩ
Active Power	[illegible] FΩ	[illegible] FΩ

5 Conclusion

In this paper, a novel Full Custom approach for dynamic voltage scaling that can be used as part of a power management system has been presented. This architecture was described through a FFT processor design. The power dissipation of the FFT multiplier core with dynamic voltage scaling was simulated, along with the MTCMOS SRAM as data memory. Power consumption expressions as functions of three control parameters (frequency, supply voltage and body bias voltage) have also been examined and presented in the paper.

The system was simulated using the 0.12um ST-Microelectronic dual-process CMOS technology. The Low Leakage/Ultra Low Leakage Library (LLULL) was chosen for the implementation process in order to minimise the system's leakage current. At an operating clock frequency of 150MHz, the total DVS power dissipa-

tion was found to be only 174.1uW. The implementation of DVS architecture has resulted in dynamic power reduction between 23-31% for 8-64 bits multiplier operation.

References

1. Cooley, J. W. and Tukey, J. W.: An Algorithm for the machine calculation of complex-Fourier series. *Math. Comput.*, Vol. 19 (1965) 297-301
2. Rabey, J. M., et. al.: Digital Integrated Circuits, A Design Perspective 2nd Edition.Pearson Education Inc, (2003)
3. Fitrio, D., Stojcevski, A. and Singh, J.: Subthreshold leakage current reduction techniques-for static random access memory. Smart Structures, Devices, and Systems II, Vol. 5649 (2005) 673-683
4. Horiguchi, M., Sakata, T. and Itoh, K.: Switched-Source-Impedance CMOS Circuit for-Low Standby Subthreshold Current Giga-Scale LSI's. IEEE J. Solid State Circuits, Vol. 28 (1993) 1131-1135 Kao, J. C., Antoniadis, A. D.: Transistor Sizing Issues and Tools for Multi-Threshold
5. CMOS Technology. DAC Proceedings (1997)
6. Kawaguchi, H., Nose, K. and Sakurai, T.: A Super Cut-Off CMOS (SCCMOS) Scheme for 0.5-V Supply Voltage with Pico-ampere Stand-By Current. IEEE J. Solid-State Circuits,Vol. 35 (2000) 1498-1501
7. Kawahara, T. et. al.: Subthreshold Current Reduction for Decoded-Driver by Self-Reverse Biasing. IEEE J. Solid State Circuits, Vol. 28 (1993) 1136-1144
8. Smith, S.W. The Scientist and Engineer's Guide to Digital Signal Processing. California Technical Publishing (1998)
9. Baek, J. H., Son, B. S., Jo, B. G., Sunwoo, M. H. and Oh, S. K.: A continuous flow mixed-radix FFT architecture with an in-place algorithm. (2003)
10. Lihong, J., Yonghong, G., Jouni, I. and Hannu, T.: A new VLSI-oriented FFT algorithm and implementation. (1998)
11. Melander, J., Widhe, T., Palmkvist, K., Vesterbacka, M. and Wanhammar, L.: An FFT processor based on the SIC architecture with asynchronous PE.(1996)
12. Kulkarni, G. R. and Udpikar, V.: An integrated facility for data acquisition and analysis of biomedical signals. Case studies on VEP, IVS. (1995)
13. Lin, W., Mittra, E., Berman, C., Rubin, C. and Qin, Y. X.: Measurement of ultrasound-phase velocity in trabecular bone using adaptive phase tracking. (2002)
14. Lisha, S., Minfen, S. and Chan, F. H. Y.: A method for estimating the instantaneous frequency of non-stationary heart sound signals. (2003)
15. Moller, F., Bisgaard, N. and Melanson, J.: Algorithm and architecture of a 1 V low power-hearing instrument DSP. (1999)
16. Baugh, C. and Wooley, B.: A two's complement parallel array multiplication algorithm.IEEE Trans. Comput., Vol. C-22 (1973) 1045-1047
17. Yang, S., Powell, M. D., Falsafi, B., Roy, K. and Vijaykumar, T. N.: An integrated circuit/architecture approach to reducing leakage in deep-submicron high-performance I-caches. Int. Symp. High Performance Computer Architecture (2001)
18. Powell, M., Se-Hyun, Y., Falsafi, B., Roy, K. and Vijaykumar, N.: Reducing leakage in a high-performance deep-submicron instruction cache. IEEE Transactions on Very Large Scale Integration (VLSI) Systems, Vol. 9 (2001) 77

Design of an Efficient Multiplier-Less Architecture for Multi-dimensional Convolution

Ming Z. Zhang, Hau T. Ngo, and Vijayan K. Asari

Computer Vision and Machine Intelligence Laboratory,
Department of Electrical and Computer Engineering,
Old Dominion University, Norfolk, VA 23529, USA
{mzhan002, hngox001, vasari}@odu.edu

Abstract. Design of a hardware efficient multiplier-less architecture for the computation of multi-dimensional convolution is presented in this paper. The new architecture performs computations in the logarithmic domain by utilizing novel multiplier-less $\log_2$ and inverse-$\log_2$ modules. An effective data handling strategy is developed in conjunction with the logarithmic modules to eliminate the necessity of multipliers in the architecture. The proposed approach reduces hardware resources significantly compared to other approaches while it still maintains a high degree of accuracy. The architecture is developed as a combined systolic-pipelined design that produces an output in every clock cycle after the initial latency of the system. The architecture is capable of operating with a high speed clock frequency of 99 MHz based on Xilinx's Virtex II 2v2000ff896-4 FPGA and the throughput of the system is observed as 99 MOPS (million outputs per second).

1 Introduction

Convolution is one of the many computationally intensive yet fundamental operations in digital signal processing applications which include speech processing, digital communications, digital image and video processing. Image processors can be used to perform convolution operation; however, these processors do not fully exploit the parallelism in this operation. In addition, kernel size is usually limited to a small bounded range. Dedicated hardware units are good for high speed processing and large kernel size; however, these dedicated units usually consume a large amount of hardware resources and high power consumption. It is necessary to find optimal designs to reduce hardware resources and power consumption while supporting high speed operations for real-time applications. The definition of N-dimensional convolution $O = W * I$ in general can be expressed as equation (1)

$$O(m_1, ..., m_N) = \sum_{j_1=-a_1}^{a_1} ... \sum_{j_N=-a_N}^{a_N} W(j_1, ..., j_N) \times I(m_1 - j_1, ..., m_N - j_N) \quad (1)$$

T. Srikanthan et al. (Eds.): ACSAC 2005, LNCS 3740, pp. 65–78, 2005.

where $a_i = \frac{J_i - 1}{2}$ and W is the kernel function. The computational complexity is $\left(\prod_{i=1}^{N} M_i \times \prod_{i=1}^{N} J_i\right)$. For N=2, the complexity is in the order O($M_1 \times M_2, J_1 \times J_2$), where in applications of image processing, $M_1 \times M_2$ is the dimension of I/O images and $J_1 \times J_2$ is the size of the kernel. For instance, in a video processing application, if the frame size is 1024×1024 and the kernel size is 10×10, more than 3 Giga-Operations-Per-Second (GOPS) are required to support real-time processing rate of 30 frames per second.

Many researchers have studied and presented different techniques and designs to support real-time computation of convolution operation. Various algorithms are presented in [1] and these algorithms focus on reducing the number of multiplications and floating operations. These algorithms are more suitable for general purpose processors or DSP processors. Examples of existing 2-D convolver chips for convolution operation depend on the multipliers in the processing element include HSP48901 and HSP48908 [2]. Other researchers presented techniques and designs which rely on the multipliers in the processing elements that are connected as a systolic architecture [3,4,5,6]. Several methods are developed to reduce the computational complexity of convolution operations such as serializing the procedure or specializing the computation in bit or gate level [7,8,9]. One of the common approaches for fast computation of convolution operation is to integrate a specialized FPGA with a DSP processor [10,11]. These methods are usually simpler and more effective than using multiprocessor approaches [12,13]. Some researchers have presented multiplier-less implementations of convolution in more restricted form [14,15,16,17]. Such implementations rely on optimization of filters with constant coefficients in which the optimization process requires the filter coefficients to be power of two to simplify the hardware structures. It is therefore inflexible for general purpose processing. In this paper, we propose a very efficient architecture for implementation of multiplier-less filters while permitting dynamic change of arbitrary kernel coefficients.

2 Theory of Log-Based Convolution Operation

The basic concept of reducing the computational complexity for multiplier-less design for multi-dimensional convolution with arbitrary kernel coefficients is based on operations in logarithmic domain in which multiplications are transformed to additions. Therefore, the key point in the theory is to convert the linear scale data into logarithmic base-two as the data is fed into the system in real time. In this section the fast approximation method for $\log_2$ and inverse-$\log_2$ is discussed in detail. For now, we assume that both $\log_2(\bullet)$ and $\log_2^{-1}(\bullet)$ operators are readily available in order to proceed with the convolution operation. The logarithmic data is then added with the kernel coefficients which are already converted in logarithmic base-two upon initialization. The actual results calculated by taking the inverse-$\log_2$ of the sums to convert back to linear scale inside the processing elements (PEs). A more detailed description of the PE design is covered in section 3. The overall output of a 1-D convolution operation, which

is available at the output of last PE, is computed by successively accumulating the partial results along a series of PEs. Hence, for K-point 1-D filters, only one $\log_2$ and K inverse-$\log_2$ computations are needed to replace K signed multipliers that are used in the conventional approach.

The same concept can be applied to multi-dimensional convolutions. The only difference is that the output from each array of PEs may require appropriate delays such that its partial results can be considered by other subsequent arrays of PEs for further processing. For an N-dimensional filter, the basic building block consists of multiple N-1 dimensional filters in successive filtering. In this paper, only 2-D convolutions are explained in complete details since one-dimensional filters are too trivial and 2-D convolutions are more common in digital image/video processing applications. The procedure also applies to higher dimensional filters. It is important to note that we are not restricting the filter coefficients to specific values or range as some of the earlier research works. For R-bit filter coefficients, each PE can have any value out of 2^R combinations.

2.1 $\mathbf{Log_2}$ and Inverse-$\mathbf{Log_2}$ Approximations

Most computational costs are dependent on the number of multiplication and division operations in the application. As $\log_2$ and inverse-$\log_2$ modules are the most critical components in reducing the computational complexity and hardware resources, it is absolutely essential to obtain efficient approximation technique in conjunction with effective data handling strategy. Although only log and inverse-log of base-two are discussed, the approximations can be applied to any base by a simple multiplication with a scaling constant. Mathematically, the complexity of $\log_2$ and inverse-$\log_2$ is more costly than multipliers if conventional approaches are used. The algorithm presented in this section utilizes binary numeric system to logically compute $\log_2$ and inverse-$\log_2$ values in a very efficient fashion. Given the definition of convolution, $O = W * I$, the one-dimensional space convolution with $\log_2$ and inverse-$\log_2$ operators can be derived as:

$$O(m) = \sum_{j=-(N-1)/2}^{(N-1)/2} W(j)I\,(m-j) = \sum_{j=-(N-1)/2}^{(N-1)/2} 2^{\log_2(W(j))+\log_2(I(m-j)} \tag{2}$$

Function $O(m)$ is now approximated to significantly reduce computational complexity:

$$\begin{aligned} O(m) &= \sum_{j=-(N-1)/2}^{(N-1)/2} 2^{V(m,j)}, 0 \le m \le M-1 \\ &\cong \sum_{j=-(N-1)/2}^{(N-1)/2} 2^{\lfloor V \rfloor} + 2^{\lfloor V \rfloor} \cdot (V - \lfloor V \rfloor) \\ &= \sum_{j=-(N-1)/2}^{(N-1)/2} \{1 \ll \lfloor V \rfloor\} + \{(V - \lfloor V \rfloor) \ll \lfloor V \rfloor\} \\ &= \sum_{j=-(N-1)/2}^{(N-1)/2} (V - \lfloor V-1 \rfloor) \ll \lfloor V \rfloor \end{aligned} \tag{3}$$

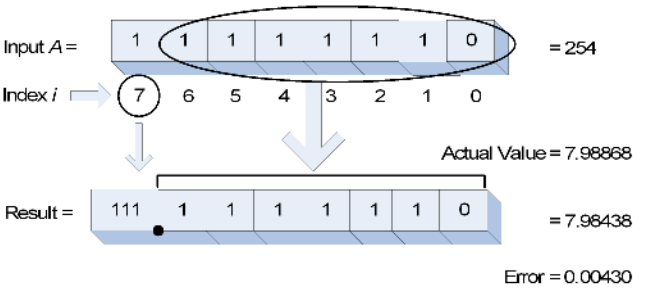


Fig. 1. An example to illustrate the concept of $\log_2$ approximation method

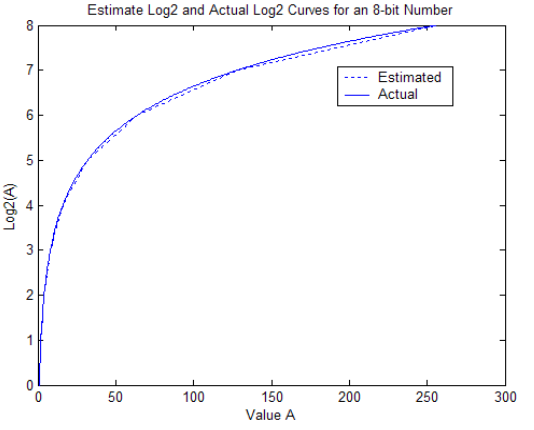


Fig. 2. Actual curve and estimated curve of $\log_2(A)$ obtained from approximation technique

where $V(m,j) = \log_2(W(j)) + \log_2(I(m-j))$, $\lfloor\rfloor$, and $\ll$ denote floor and shift operators, respectively. Note that we have derived an expression of 1-D convolution with all multiplications and divisions completely in terms of logical shift operations while maintaining the generality of functions $W(i)$ and $I(m)$. The same concept can be generalized to multi-dimensional convolutions. The $\log_2$ operator is similar to $\log_2^{-1}(L) \cong (L - \lfloor L-1 \rfloor) \ll \lfloor L \rfloor$ and can also be performed by logical operations only.

The following example is provided to illustrate the concept of estimating the value of $\log_2$. Given a positive integer A in binary form, the logical computation for $\log_2$ is achieved by determining the index value of the most significant bit (MSB) being '1' and the fraction. The index locating concept is illustrated in Fig. 1 with an 8-bit integer. In binary, every bit location corresponds to the linear value 2^i, where i is the index of the bit position. Hence integer part of $\log_2$ is extracted directly from the index value with MSB equal to one. The remaining bits after index value form the fraction of $\log_2$. For example, for an 8-bit value A=254, the index will be 7 and the approximated $\log_2$ will be 7.98438 in decimal whereas the exact value of $\log_2(A)$ is 7.98868, and hence the error is 0.00430 as illustrated in Fig. 1.

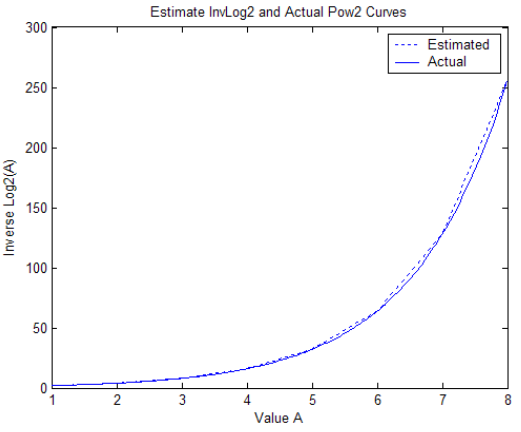


Fig. 3. Actual curve and estimated curve of inverse-$\log_2(A)$ obtained the estimation method

This logic is also true for numbers less than 1. The only difference is its indexing mechanism, where the indices range from R-1 down to -R which is represented with R-bit integer and R-bit fraction (total of $2R$ bits). The logical calculation for inverse-$\log_2$ is exactly the reverse process of the $\log_2$ approximation technique just described. The approximation error is minor and is negligible for multi-dimensional convolution in general for most applications. Fig. 2 shows the estimated curve obtained from the approximation method and the actual curve for the $\log_2$ of an 8-bit integer A. The approximation method for inverse-$\log_2$ implements the reverse-procedure of the $\log_2$ approximation where the integer part of the input number is interpreted as the index for the MSB being '1' in the result. Fig. 3 shows the estimated inverse-$\log_2$ curve and the actual inverse-$\log_2$ curve for an 8-bit input number A.

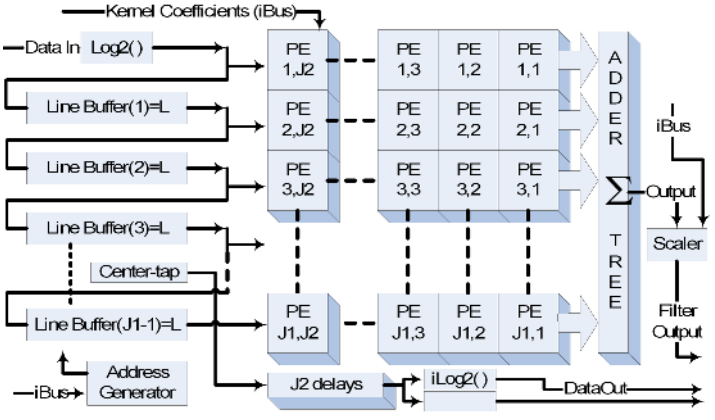


Fig. 4. Block diagram of 2-D convolution architecture

3 Architecture for Log-Based 2-D Convolution

Since the concept and computational procedure for multi-dimensional convolutions are the same as those of 2-D convolution, only the architecture design for 2-D convolution is discussed in this section. The overview of the architecture for 2-D convolution is shown in Fig. 4. As the input data being fed into the system, it is converted into $\log_2$ scale through the approximation module. For an R-bit input data, the integer portion of data in $\log_2$ scale occupies $\log_2(R)$ bits. The remaining (R - $\log_2(R)$) bits are considered as the fractional part. The $\log_2$ scaled data is then passed through J_1-1 line buffers or delay lines which is implemented with dual port RAM (DPRAM). The line buffers are managed by an address generator. The outputs of line buffers are simultaneously fed into arrays of PEs on each row and the partial outputs of the PEs along each row are successively accumulated from J_2 to 1. The outputs of the PE arrays at the last column are summed by adder tree, which can be pipelined to improve overall system speed. This filter output may be scaled by a constant in the scaler module which is simply a multiplication operation of the output with a scaled constant. The scaler module is necessary when one needs to de-normalize the kernel coefficients. For example, for the Laplacian kernel, the magnitude of coefficient at the center of the mask is usually greater than one even though the values of the entire mask are summed up to zero. Another simple but very useful structure incorporated in this architecture is the inter-chained bus (iBus) which is a chain of flip-flops where the system can be initialized without using the mapping of I/O registers.

3.1 Processing Elements (PEs)

The most crucial part of the entire architecture design process of a convolution system is the design of PEs. Often, designs of hardware architecture with simple but efficient structures are the most desired approaches. Architecture of the PEs is shown in Fig. 5 to illustrate the design of the PEs. The "Coeff In" bus is part of "iBus" where the kernel coefficients are clocked into the coefficient registers and they are propagated through the chain of registers, which is accomplished by simply feeding the output of the coefficient register in current PE to the input of coefficient register in the next PE. "Data In" bus on each PE array carries the $\log_2$ scaled data from the output of each line buffer. This data is added with the kernel coefficients. Summation of "Data In" with "Coeff", which are in log scale at all nodes is the equivalent results of multiplications in linear scale. Hence, inverse-$\log_2$ operations bring back the results to linear scale where these partial results can be accumulated in series of PEs within each PE array. The sign bit of coefficient register indicates whether addition or subtraction should be performed along accumulation line. The interface structure of adjacent PEs which is used to interact with other PEs is also shown in Fig. 5.

The accumulator output "ACCout" at the end of each PE array is fed to adder tree where the overall output of the system is calculated. The hardware components of each PE include two adders, two registers, and one inverse-$\log_2$ which is designed with an efficient approximation module. The architectures of

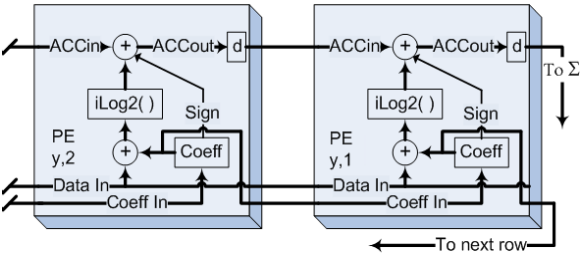


Fig. 5. Architecture of processing elements

$\log_2$ and inverse-$\log_2$ rely on finding the index of the most significant '1' bit in a binary number. Since the approximation mechanism used in this design is a logical operation instead of arithmetical calculation, it is feasible to estimate $\log_2$ and inverse-$\log_2$ within single clock cycle while still achieving high speed operation without utilizing specialized hardware or additional pipelining resources. The architectures for $\log_2$ and inverse-$\log_2$ are quite simple yet very fast and very efficient in terms of speed and hardware resources.

3.2 Log_2 Architecture

The $\log_2$ architecture consists of mainly the R-bit standard priority encoder and a modified barrel shifter (MBS). The general architectural design for $\log_2$ is shown in Fig. 6. The priority encoder provides the index output based on the

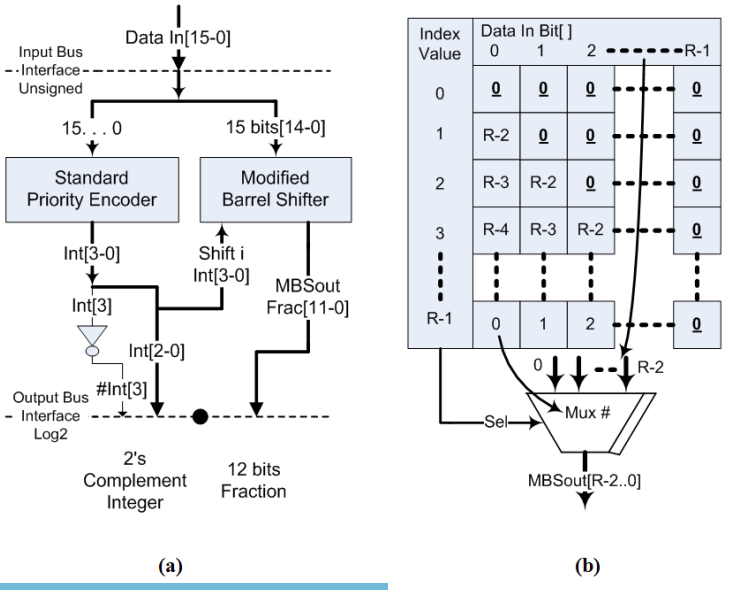


Fig. 6. (a) Architecture of $\log_2$, (b) Mapping of multiplexers in MBS

logic '1' of the highest bit in the input value. As indicated in Fig. 6a where R equals 16, the input of priority encoder is capable of encoding any 16-bit real number. If the input value is strictly a positive integer, the index output maps directly to the integer portion of $\log_2$ scale, binary 0000 to 1111 in this example. The infinity is bounded to index 0 as it is the logical function of priority encoder and that there is no need of defining $\log_2(0) = -\infty$ for practical convolutions in general. If the input value has both integer and fractional parts, the MSB of the index on the output of priority encoder is inverted to determine the actual integer part of $\log_2$ scale. For example, the index value is now mapped to [7, -8] instead of [15, 0] integer input value. Index 0 now corresponds to -8 in 2's complement. For the same reason, $\log(0) = -\infty$ is bounded to -8.

The fractional bits are extracted with modified barrel shifter. It is composed of R-1 R-to-1 multiplexers at the most. The logical functional view for mapping the set of multiplexers is that given the index, it always shifts the bit stream at the index position to be the first bit at its output. In standard barrel shifter, the output can be linearly or circularly shifted by i positions from index 0 to R-1; however, the modified barrel shifters in both $\log_2$ and inverse-$\log_2$ exhibit the reverse mapping. The mapping of R-1 multiplexers is indicated in Fig. 6b. The index value along the vertical axis represents the index that specifies i shifts. It is directly connected to the select lines of multiplexers. So for binary combination of i shifts, the corresponding input i is enabled. The outputs of multiplexers are one-to-one mapping to the R -1 bit output bus. The index on the horizontal axis represents the bit value of the input at corresponding bit location. The values within the horizontal and vertical grid specify the multiplexer numbers where the corresponding bit values of the input are mapped to. For example, with the index value of 3, bit values at locations 0 to R-1 of the input are mapped to the third set of inputs of multiplexer numbered R-4 to 0. The third set of inputs of the multiplexers outside the mapping bit range of the input is padded with zeros for simplicity. The net number of inputs of the multiplexers can be reduced by a half when the architecture of MBS is optimized, eliminating the zero-padded inputs. The fraction on the output of MBS occupies R-$\log_2$(R) bits with the fixed point $\log_2(R)$ bits down from the MSB. Note that the whole fraction can be preserved; however it is truncated to R-$\log_2(R)$ bits so the integer and fraction add up to the same bus size as the input.

The maximum propagation delay of the $\log_2$ architecture is computed based on the critical path of the combinational network in priority encoder and modified barrel shifter where the modified barrel shifter depends on the index from priority encoder to perform i shifts. Note that the arrangement of multiplexers is completely in parallel such that the overall latency comprises a single multiplexer. The depth of propagation delay is significantly less compared to non-pipelined conventional multipliers. It implies that the architecture can provide very high speed operations.

3.3 Inverse-Log$_2$ Architecture

Structural mapping of inverse-log$_2$ is the reverse one of log$_2$, as illustrated in Fig. 7b. The inverse-log$_2$ architecture is simpler than log$_2$ architecture since it is not necessary to have the decoder to undo the priority encoding where the integer part serves as i shifts to the reverse of the modified barrel shifter (RMBS). The inverter is not needed for the inverse-log$_2$ architecture shown in Fig. 7a for log$_2$ scaled inputs greater than or equal to zero. Note that negative values of log$_2$ scale indicate the inverse-log$_2$ result in linear scale should be a fraction.

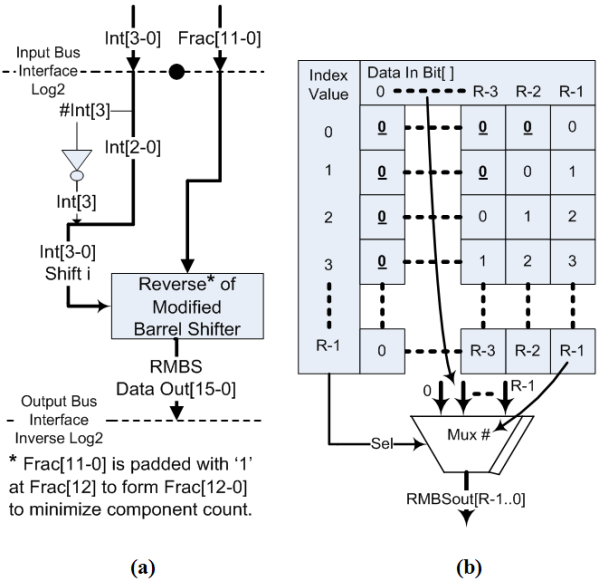


Fig. 7. (a) Architecture of inverse-log$_2$, (b) Mapping of multiplexers in RMBS

For applications where such small numbers are insignificant, the hardware resource can be reduced by half for the conversion of signed inverse-log$_2$ scale to linear scale. Another important point is that the fraction bits fed to the reverse of the modified barrel shifter should be padded with logic '1' at the MSB such that the magnitude of index can be restored in binary. It is the equivalence to performing the OR operation between the decoded bit and the unpadded fraction bits if the decoder was included in the architecture to form the exact reverse of log$_2$ architecture. The difference in processing time can be quite significant with such a minor modification. The operating frequency of inverse-log$_2$ architecture is estimated to be twice that of the log$_2$ architecture as the propagation delay of the critical path is reduced to half.

4 Simulation Results

In this section, the setup of simulation parameters is examined. Two types of filters are applied to evaluate the performance of designed architecture as well as the accuracy of the proposed method. In the first set of the simulations, we looked at a common operation in image processing applications which uses a Laplacian kernel to detect edges in grayscale images. The second operation which involves 2-D convolution is the noise removal in image processing applications. For the second set of experimental simulations, we used a general Gaussian kernel for the filter. The hardware simulation of 2-D convolution was tested on a set of JPEG images to determine the accuracy of the results produced by hardware architecture. Each image was converted to grayscale of 8-bit resolution and fed into the architecture pixel by pixel. The constraint of data values in the simulation of architecture was strictly checked and validated at all time. The $\log_2$ and inverse-$\log_2$ architectures were restricted to 8-bit resolution (3 bits for integer and 5 bits for fraction). To optimize the hardware resource, the kernel coefficients were normalized to have maximum magnitude of one in linear scale (the actual values are in $\log_2$ scale and these coefficients are stored in coefficient registers). The inverse-$\log_2$ module inside each PE was set to 16 bit resolution (the output has 8 bits integer and 8 bits fraction) for maximum precision.

4.1 Edge Detection by Laplacian Kernel

Fig. 8a shows the Laplacian kernel with the magnitude of coefficients normalized to [0, 1]. Fig. 8b is the grayscale test image. The result of applying edge detection mask to 2-D convolution by Matlab function is shown in Fig. 8c. The intermediate calculations involved are double precision. The resulting image from hardware simulation is shown in Fig. 8d. It is clear that the approximated image obtained from the architecture simulation is fairly close to the actual image filtered with Matlab function. A scaled/de-normalized version of the simulation result is shown in Fig. 8e. There are some minor effects resulted from padding

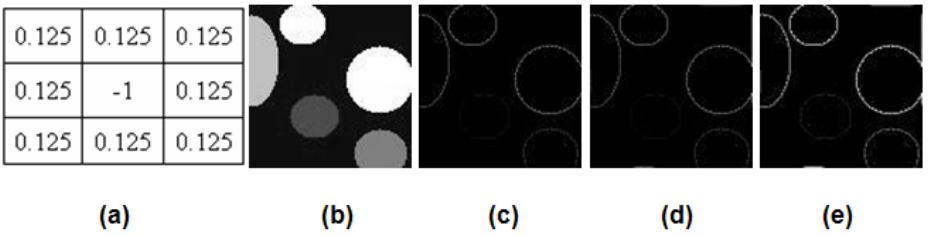


Fig. 8. Edge detection with Laplacian kernel: (a) normalized Laplacian kernel, (b) grayscale input image, (c) 2-D convolution result by Matlab function, (d) 2-D convolution result by hardware simulation, (e) hardware simulation result scaled by 2

procedure along the borders of the image. It is because the architecture pads the data on the opposite borders; however, the effect is insignificant in general. Techniques which can be used to eliminate the padding effect include padding with zero, padding with symmetrical values, etc. [18].

4.2 Noise Filtering by Gaussian Kernel

The corrupted grayscale image of Fig. 8b is shown in Fig. 9a. The Gaussian white noise with mean m=0, and variance σ^2=0.05 was applied to the grayscale image. A 10×10 Gaussian filter with standard deviation was quantized according to the architecture and it was convolved with the corrupted image. The results by Matlab function and hardware simulation are shown in Fig. 9b and 9c respectively. In all experiments, it was observed that the error is negligible; hence, the inverse-$\log_2$ may be optimized to 8 bits and ignoring the fractions, in addition to eliminating padded zeros of architectures shown in Fig.s 6 and 7.

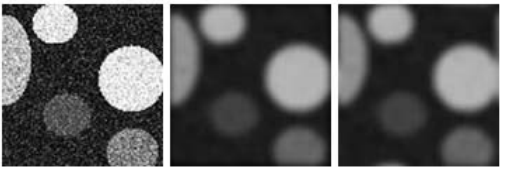


Fig. 9. Smoothening of image corrupted by Gaussian white noise: (a) grayscale of input image corrupted by Gaussian white noise with m=0, and σ^2=0.05, (b) image filtered by Matlab function, (c) image filtered by the proposed hardware

4.3 Performance Analysis

The main components of architecture presented in section 3 are $\log_2$, inverse-$\log_2$, and PE. These components have been simulated on Xilinx's Integrated Software Environment (ISE). The performances and resource utilization of $\log_2$/inverse-$\log_2$ architectures are shown in Tables 1 and 2. The maximum operating frequencies are 205 MHz, 121 MHz, and 100 MHz for $\log_2$ module with 8, 16, and 32 bit resolutions respectively. Similarly, the inverse-$\log_2$ module is capable of operating with 305 MHz, 235 MHz, and 212 MHz for 8, 16, and 32 bit resolutions respectively. As expected, the performance of inverse-$\log_2$ architecture is better than that of $\log_2$ architecture (approximately 2 times faster).

The maximum operating frequency is shown in Table 3 where each PE is capable of operating with 114 MHz, 99 MHz, and 81 MHz for 8, 16, and 32 bit resolutions respectively. For 16-bit PE, the complete implementation of non-optimized convolution architecture is estimated to have clock frequency close to 99 MHz. Since the architecture is designed as a pipelined-systolic system, it can produce an output every cycle after the initial latency of the system

Table 1. Performance and hardware utilization of $\log_2$ architecture with various resolutions

Description	Resolution		
	8	16	32
CLBs	11	43	166
LUTs	19	76	289
F_{max}(MHz)	205	121.5	99.99

Table 2. Performance and hardware utilization of inverse-$\log_2$ architecture with various resolutions

Description	Resolution		
	8	16	32
CLBs	12	44	164
LUTs	19	70	268
F_{max}(MHz)	305.6	235.4	212.2

Table 3. Performance and hardware utilization of PE architecture with various resolutions

Description	Resolution		
	8	16	32
CLBs	19	60	196
LUTs	34	102	332
F_{max}(MHz)	114.5	99.2	81.1

Table 4. Comparison of hardware resources and performance with other 2-D convolution implementations

Kernel 3× 3	HSP48901 [2]	Method in [12]	Proposed design
$F_{clk\ max}$(MHz)	30	125	99
Gate count	13,594	3,893	4,212*
Row buffers	Off chip	fetched	On chip
Data $rate_{max}$ (Mpixels/sec.)	30	15.6	99

* Number of logic gates is obtained based on fully optimized architecture with a 9-bit signed integer and 8-bit fraction resolution

which is insignificant compared to the overall operation time. The architecture is therefore capable of sustaining a throughput rate of 99 Mega-outputs per second which is very suitable for real-time image or signal processing applications. With 1024×1024 frame size in video processing and 10×10 kernel dimension, the architecture is capable of performing at 94.6 frames per second or equivalently 9.9GOPS. A comparison of hardware resources and the performance with 3 other systems for 2-D convolution is shown in Table 4.

5 Conclusion

A new architecture for computing multi-dimensional convolution without using multipliers has been proposed in this paper. The approach utilized approxima-

tion techniques to efficiently estimate $\log_2$ and inverse-$\log_2$ for supporting convolution operation in logarithmic domain. The approximation architectures can process over 100 million and 200 million calculations for 16-bit $\log_2$ and inverse-$\log_2$ operations per second respectively. An effective and intelligent data handling strategy was developed to support the approximation architecture which resulted in the elimination of the need for multipliers in the convolution operation. The architecture design for 2-D convolution is capable of maintaining a throughput rate of 99 Mega-outputs per second for 16-bit PE resolution in Xilinx's Virtex II 2v2000ff896-4 FPGA at a clock frequency of 99 MHz.

References

1. A. F. Breitzman, "Automatic Derivation and Implementation of Fast Convolution Algorithms," Thesis, Drexel University, 2003.
2. HARRIS semiconductor Inc., Digital Signal Processing, 1994.
3. H. T. Kung, L. M. Ruane, and D. W. L. Yen, "A Two-Level Pipelined Systolic Array for Multidimensional Convolution," Image and Vision Computing, vol. 1, no. 1, pp. 30-36, Feb. 1983.
4. A. Wong, "A New Scalable Systolic Array Processor Architecture for Discrete Convolu-tion," Thesis, University of Kentucky, 2003.
5. A. E. Nelson "Implementation of Image Processing Algorithms on FPGA Hardware," Thesis, Vanderbilt University, 2000.
6. C. Chou, S. Mohanakrishnan, J. Evans, "FPGA Implementation of Digital Filters," in Proc. ICSPAT, pp. 80-88, 1993.
7. H. M. Chang and M. H. Sunwoo, "An Efficient Programmable 2-D Convolver Chip," Proc. Of the 1998 IEEE Intl. Symp. on Circuits and Systems, ISCAS, part 2, pp. 429-432, May 1998.
8. K. Chen, "Bit-Serial Realizations of a Class of Nonlinear Filters Based on Positive Boo-lean Functions," IEEE Trans. On Circuits and Systems, vol. 36, no. 6, pp. 785-794, June 1989.
9. M. H. Sunwoo, S. K. Oh, "A Multiplierless 2-D Convolver Chip for Real-Time Image Processing," Journal of VLSI Signal Processing, vol. 38, no. 1, pp. 63-71, 2004.
10. B. Bosi and G. Bois, "Reconfigurable Pipelined 2-D Convolvers for fast Digital Signal Processing," IEEE Trans. On Very Large Scale Systems, vol 7, no. 3, pp. 299-308, Sept. 1999.
11. M. Moore, "A DSP-Based Real-Time Image Processing System," Proc. of the 6th Intl. Conf. on Signal Processing Applications and Technology, Boston, MA, pp. 1042-1046, August 1995.
12. J. H. Kim and W. E. Alexander, "A Multiprocessor Architecture for 2-D Digital Filters," IEEE Trans. On Computer, vol. C-36, pp. 876-884, July 1987.
13. M. Y. Dabbagh and W. E. Alexander, "Multiprocessor Implementation of 2-D Denomina-tor-Separable Digital Filters for Real-Time Processing," IEEE Trans. on Acoustics, Speech, and Signal Processing, vol. ASSP-37, pp. 872-881, June 1989.
14. K. Wiatr, E. Jamro, "Constant Coefficient Multiplication in FPGA Structures," Proc. of the 26th Euriomicro Conference, Maastricht, The Netherlands, vol. 1, pp. 252-259, Sept. 5-7, 2000.

15. W. J. Oh, Y. H. Lee, "Implementation of Programmable Multiplierless FIR Filters with Power-of-Two Coefficients," IEEE Trans. on Circuits and Systems - Part - Fundamental Theory and Applications, vol.42, no.8, pp. 553-556, 1995.
16. S. Samadi, H. Iwakura, and A. Nishihara, "Multiplierless and Hierarchical Structures for Maximally Flat Half-Band FIR Filters," IEEE Trans. Circuits Syst.-II: Analog and Digi-tal Signal Process, vol. 46, pp. 1225-1230, 1999.
17. J. Yli-kaakinen and T. Saramaki, "A Systematic Algorithm for the Design of Multiplier-less FIR Filters," Proc. IEEE Int. Symp. Circuits Syst., Sydney, Australia, vol. II, pp. 185-188, May 6-9, 2001.
18. M. Z. Zhang, H. T. Ngo, A. R. Livingston, and K. V. Asari, "An Efficient VLSI Architec-ture for 2-D Convolution with Quadrant Symmetric Kernels," IEEE Computer Society Proc. of the Intl. Symp. on VLSI - ISVLSI 2005, Tampa, Florida, pp. 303-304, May 11 - 12, 2005.

A Pipelined Hardware Architecture for Motion Estimation of H.264/AVC

Su-Jin Lee, Cheong-Ghil Kim, and Shin-Dug Kim

Dept. of Computer Science, Yonsei University,
134, Sinchon-Dong, Seodaemun-Gu, Seoul 120-749, Korea
{haegy, tetons, sdkim}@yonsei.ac.kr

Abstract. The variable block size motion estimation (VBSME) presented in the video coding standard H.264/AVC significantly improves coding efficiency, but it requires much more considerable computational complexity than motion estimation using fixed macroblocks. To solve this problem, this paper proposes a pipelined hardware architecture for full-search VBSME aiming for high performance, simple structure, and small controls. Our architecture consists of 1-D arrays with 64 processing elements, an adder tree to produce motion vectors (MVs) for variable block sizes, and comparators to determine the minimum of MVs. This can produce all 41 MVs for variable blocks of one macroblock in the same clock cycles to other conventional 1-D arrays of 64 PEs. In addition, this can be easily controlled by a 2-bit counter. Implementation results show that our architecture can estimate MVs in CIF video sequence at a rate of 106 frames/s for the 32×32 search range.

Keywords: Motion estimation, variable block size, full search, array architecture, H.264/AVC, and video coding.

1 Introduction

The ITU-T and ISO/IEC draft H.264/AVC video coding standard [1] achieves significantly improved coding performance over existing video coding standards such as H.263 and MPEG-4 due to new added features. Among such features, motion estimation (ME) with variable block sizes especially contributes the video compression ratio, because ME of H.264/AVC minimizes differences between video frames by dividing a 16×16 macroblock into smaller blocks with variable sizes. This improves not only the coding efficiency but also the quality of video frames with small object movements. However, it requires much more considerable computational complexity than ME of previous coding standards using fixed 16×16 macroblocks. Moreover, if full-search ME is used, ME consumes most of the total encoding time [2]. This problem can be solved by designing a dedicated hardware for ME, particularly in video encoding systems for high performance and real time processing.

This paper proposes a pipelined hardware architecture for full-search variable block size motion estimation (FSVBSME) aiming for high performance, simple struc-

T. Srikanthan et al. (Eds.): ACSAC 2005, LNCS 3740, pp. 79 – 89, 2005.

ture, and small controls. Though ME architectures for FSVBSME are presented previously [3], [4], [5], [6], they require many resources or complex controls. Our architecture consists of 1-D arrays with a total of 64 processing elements (PEs) and an adder tree to produce motion vectors (MVs) for variable blocks. In other words, four 1-D arrays with 16 PEs compute the sum of absolute differences (SAD) for 4x4 blocks, the smallest block of proposed variable blocks. Then the adder tree produces the SADs for larger blocks by combining the SADs for 4×4 blocks. In order to perform this process efficiently, a pipeline technique is used in our architecture. Thereby, the proposed architecture can produce 41 MVs for all variable blocks of one macroblock within the same clock cycles that a traditional 1-D array with 64 PEs requires in fixed block size ME. In addition, since we do not change each PE of arrays but attach only the adder tree below arrays, the proposed architecture has a very simple structure. And this can be easily controlled by a 2-bit counter to generate MVs for variable blocks, because all operations are executed at specified clock cycle.

The rest of this paper is organized as follows. Section 2 introduces the ME algorithm of H.264/AVC. In Section 3, the proposed architecture and its details are presented, and Section 4 shows the implementation result. Finally, we conclude in Section 5.

2 H.264/AVC Motion Estimation

Motion estimation originally means a search scheme which tries to find the best matching position of a 16×16 macroblock (MB) of the current frame with any 16×16 block within a predetermined or adaptive search range in the previous frame. The matching position relative to the original position is described by a motion vector, which is included in the bitstream instead of a matched MB. However, as a ME algorithm by itself is not standardized, there exist several variations to execute ME efficiently. Among them, the full-search block matching algorithm can find the best block match, since it compares a current MB with all 16×16 blocks in a search range. Moreover, it is most popular in hardware design for ME because of its regularity. In the full search block matching, the criterion to determine the best MV is the SAD value similarly to other ME algorithms. The equations (1) and (2) describe operations to compute a MV of a N×N MB [7].

$$SAD(dx,dy) = \sum_{m=x}^{x+N-1} \sum_{n=y}^{y+N-1} | F_k(m,n) - F_{k-1}(m+dx, n+dy) | . \quad (1)$$

$$\overrightarrow{MV} = (MV_x, MV_y) = \min_{(dx,dy)\in R^2} SAD(dx,dy). \quad (2)$$

The previous video coding standards adopt the ME algorithm with the fixed block size of 16×16 pixels and, going a step further, 8×8 pixels. But H.264/AVC formally presents the ME algorithm for 7 kinds of variable block sizes, which can be divided into two groups, namely macroblock partitions of 16×16, 16×8, 8×16, and 8×8 as shown in Fig. 1 and sub-macroblock partitions of 8×4, 4×8, and 4×4 as in Fig. 2. A MB has one macroblock partition mode, and each block split by a selected mode

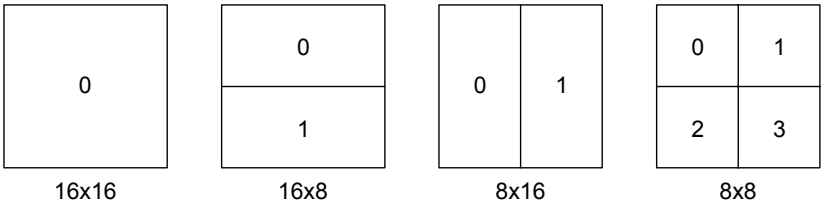


Fig. 1. Macroblock partitions

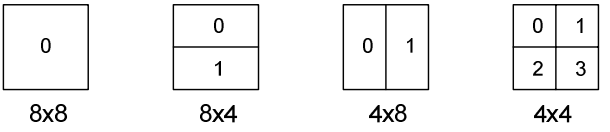


Fig. 2. Sub-macroblock partitions

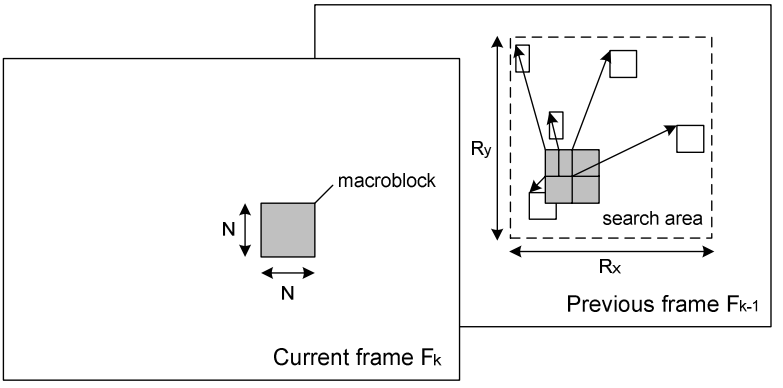


Fig. 3. Block matching motion estimation algorithm with variable block sizes

holds its own MV as depicted in Fig. 3. If the 8×8 mode is chosen as the best match, each of four 8×8 sub-macroblocks is split in a further 4 ways in Fig. 2 [8].

The full-search block matching concept can be applied to the variable block size ME algorithm to determine a partition mode and each MV. This assures the best block match in the same way as the full-search fixed block size ME, but it needs much more complicated operations since MVs should be decided for all partitions. In other words, ME of H.264/AVC ought to produce 41 MVs which is the sum of 1 MV for the 16×16 block, 2 MVs for 16×8, 2 MVs for 8×16, 4 MVs for 8×8, 8 MVs for 8×4, 8 MVs for 4×8, and 16 MVs for 4×4.

Several studies have been introduced to reduce computational complexity due to FSVBSME based on devising dedicated hardware architectures. The work in [3] uses a one dimensional (1-D) array architecture with 16 PEs where the structure of each PE is modified for FSVBSME. In [4], the reference software of H.264/AVC is altered in a hardware-friendly way and the 2-D array with 256 PEs is designed under the changed algorithm. The work in [5] describes a 2-D array architecture as [4], but it is equipped with an AMBA-AHB on-chip-bus interface. In [6], instead of a new array architecture, flexible register arrays are proposed. They offer appropriate input data, so that the computing unit can calculate motion vectors for variable blocks. All of them can support the seven kinds of variable block sizes required for H.264/AVC ME. However, the architecture in [3] has very complicated PEs and needs many control logics even if it can produce all motion vectors by using only a 1-D array. The work in [4] and [5] show powerful performance due to 256 PEs, but they use many resources to save and synchronize temporary results when extracting motion vectors for variable blocks. In case of [6], since the proposed architecture targets a 1-D array with16 PEs as the computing unit, register arrays should be extended if aiming high performance and speed.

Consequently, a 1-D array with 16 PEs uses a few resources relatively but requires complicated control logics, on the other hand, a 2-D array of 256 PEs has high performance but needs many resources to adjust SAD values of 4×4 blocks for calculating the SADs of variable block sizes. Therefore, we determine to use four 1-D arrays with a total of 64 PEs. It works more slowly than an array with 256 PEs does, but it does not demand many latches due to producing SADs of 4×4 blocks too quickly. Even though our architecture uses more resources than a 1-D array with 16 PEs, we extract four SADs for 4×4 blocks at a time from four arrays, so that using these results the proposed architecture can generate SADs for larger blocks immediately. Thus it needs not to store or load SADs temporarily and as [3] to compute SADs for larger blocks.

3 Proposed Architecture

The proposed architecture consists of three parts. The first part is the 4×4 SAD calculator (44Cal) which computes SADs for sixteen 4×4 blocks split in a macroblock. Fig. 4 shows 4×4 blocks of one macroblock and their combination to construct variable blocks of 8×4 to 16×16 size. Therefore, we calculate only SADs for 4×4 blocks. Then the second part, the variable block size SAD generator (VBGen), receives the SADs from 44Cal and computes SADs of larger blocks by internal adders. The results of the second part flow to the minimum generator (MinGen) of the third part which stores the minimum SAD for each block size. In this part, the SADs from VBGen are compared with the minimum SADs stored previously. Differently from full search ME for the fixed block size which has only one minimum SAD, FSVBSME needs to maintain minimums for all block sizes. In Fig. 5, it is shown how three parts are constructed.

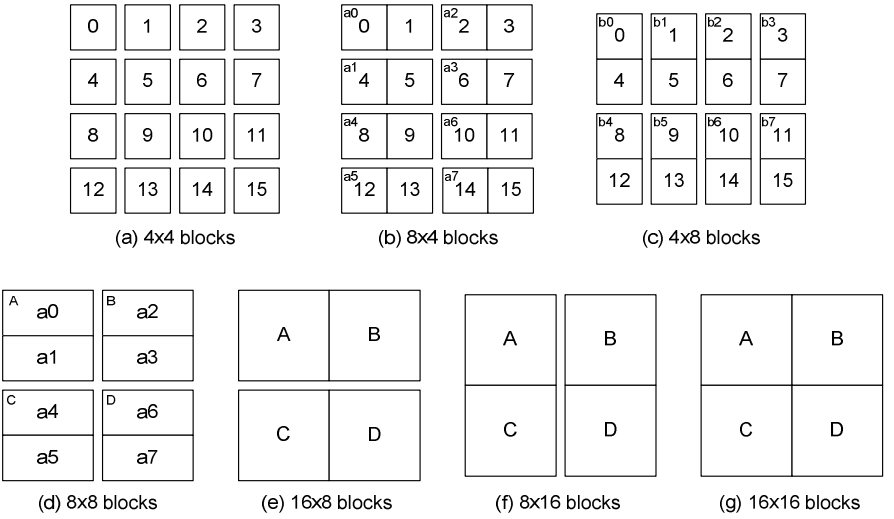


Fig. 4. Blocks of variable block sizes constructed from 4×4 blocks

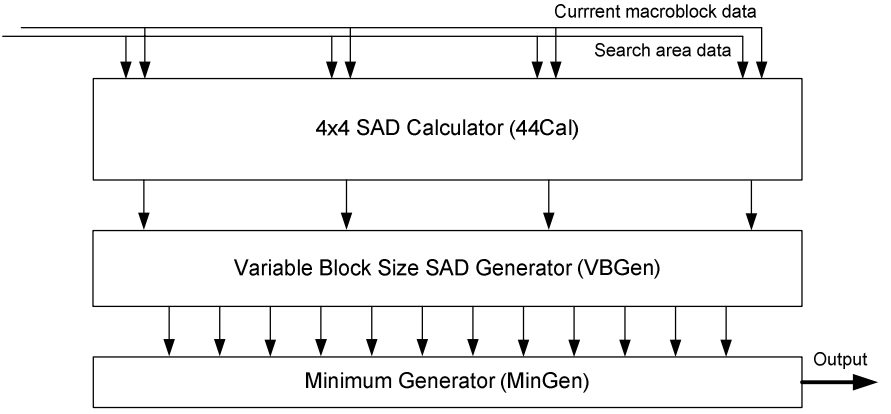


Fig. 5. Structure of the proposed architecture

3.1 4×4 SAD Calculator

The 4×4 SAD calculator is composed of four 1-D arrays as mentioned above. Each of 1-D arrays, called the basic array (BA), has 16 absolute difference units and adders to calculate the sum of AD results as presented in Fig. 6(a). c presents a pixel of the 4×4 current block and r indicates a pixel of the 4×4 reference block.

Fig. 6(b) describes the structure of 44Cal where C and R mean the 4×4 current block and the 4×4 reference block respectively. The first basic array (BA1) computes SADs for the 0th, 2nd, 8th, and 10th block, BA2 for the 1st, 3rd, 9th, and 11th block, BA3 for the 4th, 6th, 12th , and 14th block, and BA4 for the 5th, 7th, 13th, and 15th

block in Fig. 4(a). This input order is considered to build SADs of 8×4 blocks and next larger blocks in Fig. 4(b) conveniently. Since the order is very regular even if it is not sequential, we can also generate regular memory addresses to read. Four SADs of 4×4 blocks produced from 44Cal (for example, SADs of 0th, 1st, 4th, and 5th block at the first clock cycle) are transmitted to the variable block size SAD generator every clock cycle.

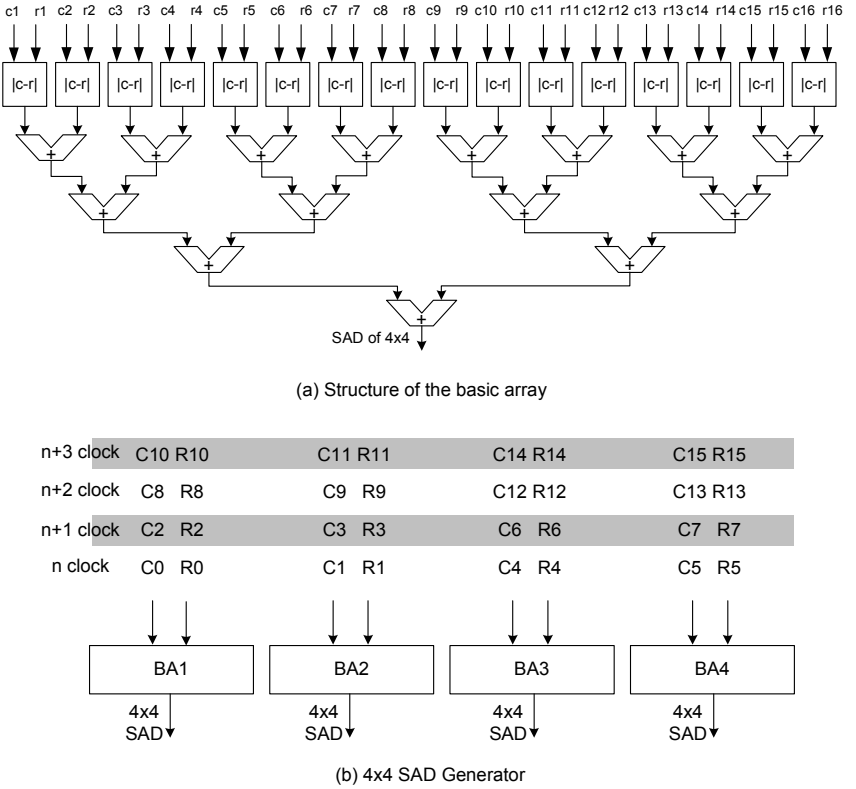


Fig. 6. Structure of the basic array and 4×4 SAD generator

3.2 Variable Block Size SAD Generator

The variable block size SAD generator is the core of the proposed architecture. It consists of adders that can be performed at the same clock speed with the arrays of 44Cal by a pipeline technique. After receiving the SADs calculated by the 44Cal at the nth clock, for the (n+1)th clock VBGen adds 4×4 SADs and makes the SADs of 8×4 and 4×8 by using the 1st, 2nd, 3rd, and 4th adder shown in Fig. 7. At the (n+2)th clock, the 5th adder computes the SADs of 8×8 blocks from 8×4 SADs. These outcomes flow into the 6th, 7th, and 8th adder to construct 16×8, 8×16, and 16×16 SADs respectively at the (n+3)th clock. In order to calculate them, the combinations of (A,B), (C,D), (A,C), (B,D), and (A,B,C,D) are required where A, B, C, and D means the SADs of 8×8 blocks as depicted in Fig. 4(d). Thus delay units (D in Fig. 7) are

added before the 6th, 7th, and 8th adder for accurate operations. Table 1 shows the results from each out line of VBGen as a block type.

By the way, as shown in Table 1, the 6th, 7th, and 8th adder inevitably generate waste results, since this architecture operates incessantly. However, the results are iterative every four clock cycle, these values can be managed by a simple control unit. In our architecture, the delay unit of the 8th adder is initiated every four clock cycle by 2-bit counter. This counter is also used for MinGen to determine where the results from 12 output lines are compared with the minimum SADs and stored every clock cycle.

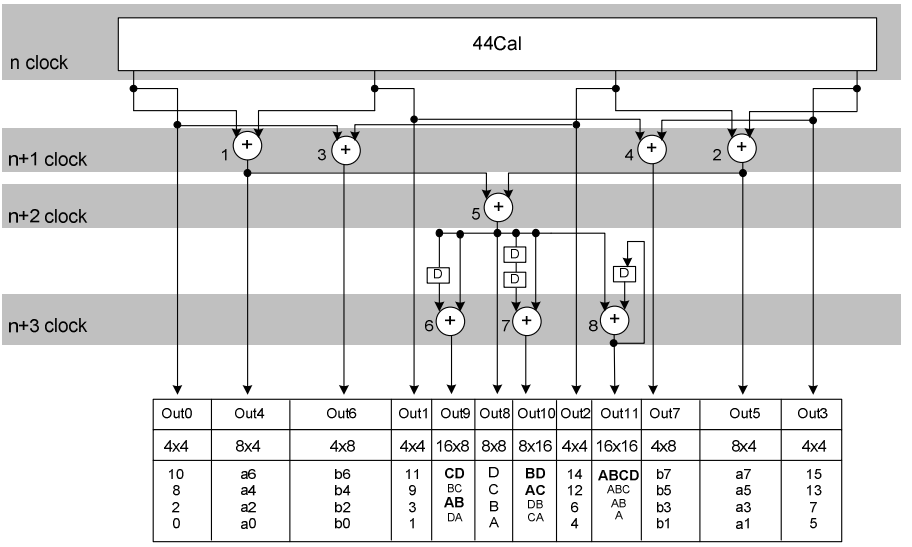


Fig. 7. Structure of the variable block size SAD generator

Table 1. Outputs of the variable block size SAD generator

	4x4	4x4	4x4	4x4	8x4	8x4	4x8	4x8	8x8	16x8	8x16	16x16
Clk	Out0	Out1	Out2	Out3	Out4	Out5	Out6	Out7	Out8	Out9	Out10	Out11
1	0	1	4	5	.	.	.	.	.	.	.	.
2	2	3	6	7	a0(0,1)	a1(4,5)	b0(0,4)	b1(1,5)	.	.	.	.
3	8	9	12	13	a2(2,3)	a3(6,7)	b2(2,6)	b3(3,7)	A(a0,a1)	.	.	.
4	10	11	14	15	a4(8,9)	a5(12,13)	b4(8,12)	b5(9,13)	B(a2,a3)	?A(x)	?A(x)	A(x)
5	0	1	4	5	a6(10,11)	a7(14,15)	b6(10,14)	b7(11,15)	C(a4,a5)	AB	?B(x)	AB(x)
6	2	3	6	7	a0(0,1)	a1(4,5)	b0(0,4)	b1(1,5)	D(a6,a7)	BC(x)	AC	ABC(x)
7	8	9	12	13	a2(2,3)	a3(6,7)	b2(2,6)	b3(3,7)	A(a0,a1)	CD	BD	ABCD
8	10	11	14	15	a4(8,9)	a5(12,13)	b4(8,12)	b5(9,13)	B(a2,a3)	DA(x)	CA(x)	A(x)
9	0	1	4	5	a6(10,11)	a7(14,15)	b6(10,14)	b7(11,15)	C(a4,a5)	AB	DB(x)	AB(x)
10	2	3	6	7	a0(0,1)	a1(4,5)	b0(0,4)	b1(1,5)	D(a6,a7)	BC(x)	AC	ABC(x)
11	8	9	12	13	a2(2,3)	a3(6,7)	b2(2,6)	b3(3,7)	A(a0,a1)	CD	BD	ABCD
...	...	...	...	...	...	...	...	...	...	...	...	...

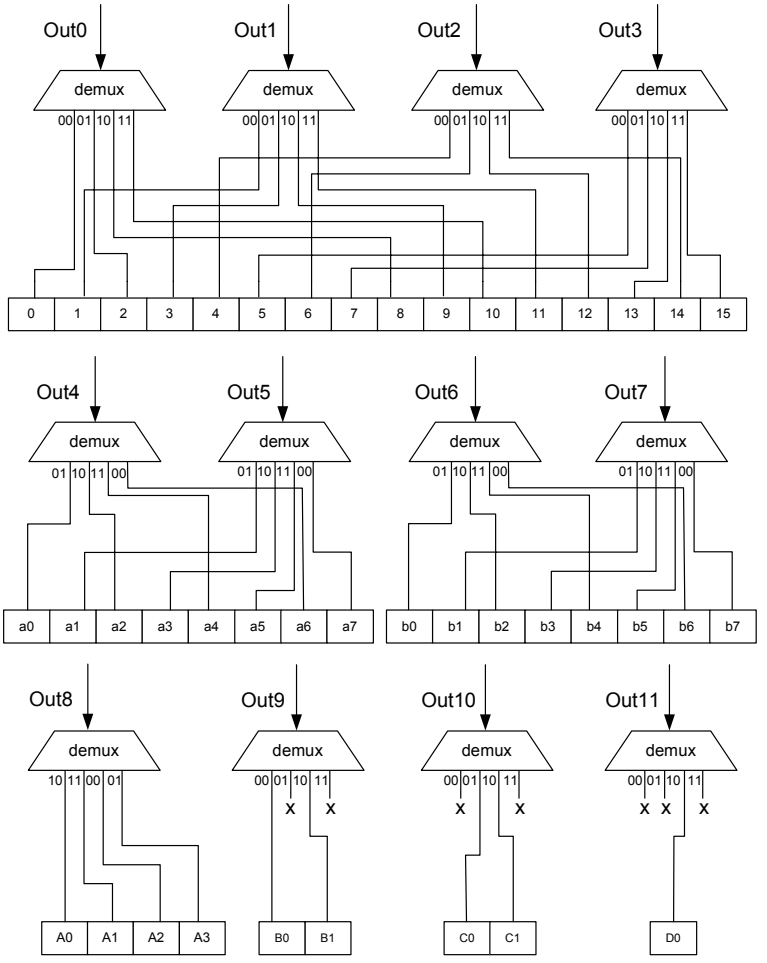


Fig. 8. Structure of the minimum generator

3.3 Minimum Generator

The minimum generator keeps the minimum SADs for each variable block. Based on the FSVBSME algorithm, the proposed architecture should compute 41 motion vectors for one macroblock, and thus demands 41 comparison units and 41 registers to store the minimums. But since the number of output lines of VBGen is just 12, we must distribute results of VBGen to the comparison units corresponding to their block sizes appropriately. This operation can be executed by the 2-bit counter mentioned above. In other words, output lines send their results to the demultiplexers connected with each output line. Demultiplexers distribute the SADs to correspond comparison units by the 2-bit counter as depicted in Fig. 8. This process plays a role to filter out waste values generated from Out9, Out10, and Out11.

As shown in Table 1, 41 SADs from a current macroblock and a reference block are repeated every 4 clock cycles. Suppose a search range is 16×16. Since the full search algorithm requires 256 block matching and our architecture can produce 41 SADs for one block match every 4 clock cycles by the pipeline technique, 256 × 4 + 3 (flush time) = 1027 cycles are required to determine 41 motion vector of one macroblock in 16×16 search range. Consequently, our architecture can compute 41 SADs for variable block sizes within the similar time to the execution time of a 1-D array with 64 PEs for fixed block size ME (1024 cycles). In our architecture, the 1-D array of 64 PEs operates as if executing fixed block size ME and the pipelined adder tree to generate SADs for variable blocks (VBGen) operates below the array by a pipeline technique. This is why the execution time is similar to that of an existing array for fixed block size ME.

4 Implementation

FPGA (Filed Programmable Gate Array) is a high-performance and economic method of IC implementation because of its short development period and flexibility. Our proposed architecture is implemented using Verilog and prototyped on a single Xilinx Spartan-3 FPGA chip. Two RAMs are used for the search window data and the current block data respectively. And for prompt execution of the array, input data flow from shift register files to 44Cal every clock cycle. The created design contains 97K gates and can operate at frequencies of up to 178MHz. Table 2 presents the specification of the described architecture.

Table 2. Specification

Algorithm	Full Search Variable Block Size ME
Number of PE	64
Frame size	QCIF(176×144), CIF(352×288), 4CIF (704×576)
Search range	16×16, 32×32
Block size	4×4, 4×8, 8×4, 8×8, 16×8, 8×16, 16×16
Frame rate (fps)	27 (4CIF & 32×32) to 1535 (QCIF & 16×16)
Target device for compilation	xc3s2000-5fg900
Supply voltage	3.3V
Gate counts	97,213
Number of I/O pads	215
Maximum frequency	178MHz

Table 3. Comparison of some FS circuits for variable block sizes

	Shen '02 [9]	Huang '03 [4]	Yap '04 [3]	This work
Number of PE	64	256	16	64
Frame size	QCIF, CIF	4CIF	QCIF, CIF, 4CIF	QCIF, CIF, 4CIF
Search range	16×16, 32×32	24x24(h), 16x16(v)	16×16, 32×32	16×16, 32×32
Block size	8×8, 16×16, 32×32	4×4, 4×8, 8×4, 8×8,16×8, 8×16,16×16	4×4, 4×8, 8×4, 8×8, 16×8,8×16, 16×16	4×4, 4×8, 8×4, 8×8, 16×8,8×16, 16×16
Frame rate (fps)	30 to 120	30	181 (CIF/16×16)	106 (CIF/32×32)
Supply voltage	2.5V / 5V	-	1.2V	3.3V
Gate counts	67k	105k	61k	97k
Frequency	60MHz	67MHz	294MHz	178MHz

The performance depicted in Table 2 means that this architecture can process a variety of frame size and search range. A brief comparison with some circuits designed for FSVBSME is given in Table 3. The architecture of Shen [9] consists of a array with 64 PEs and comparators for SADs of variable blocks. But since it is designed to accumulate SAD for a unit of 8×8 block, this architecture cannot satisfy the condition of sub-macroblock partition without structural modifications. The work of Huang [4] and our architecture is similar in point of gate counts, our architecture has higher frequency. And it is not present how to handle results for variable blocks but we can manage them by the 2-bit counter. Because the architecture of Yap[3] needs to use multiplexers programmed using look-up tables, the simple control of our architecture can be more valuable.

Actually, an accurate comparison is difficult due to the fact that those have been implemented by different methods. However, it can be found that our architecture design has reasonable gate counts and high throughput. Above all, it becomes possible by the simple control of 2-bit counters. Therefore, it is very appropriate to the H.264 encoder aiming for high-performance and real-time processing.

5 Conclusion

A new pipelined architecture for variable block size motion estimation is proposed in this paper. This architecture can process all 41 motion vectors for variable block sizes within 4 clock cycles. It needs only four arrays and very simple adder tree structure combining results of the arrays by a pipelined method. In addition, when storing SAD values and deciding the minimums of 41 motion vectors, a simple 2-bit counter can

control related operations. And it can accept various search ranges and frame sizes sufficiently. These properties imply that this architecture has good performance and high flexibility, and thus it is very suitable for design of the real-time H.264 video encoder.

References

1. ISO/IEC 14496-10: Coding of Moving Pictures and Audio (2002)
2. J. Zhang, Y. He, S. Yang, and Y. Zhong: Performance and Complexity Joint Optimization for H.264 Video Coding, Proceedings of the 2003 International Symposium on Circuits and Systems, Vol. 2. (2003) 888–891
3. S.Y. Yap and J.V. McCanny: A VLSI Architecture for Variable Block Size Video Motion Estimation, IEEE Transactions on Circuits and Systems II: Express Briefs, Vol. 51, issue 7. (2004) 384–389
4. Y.W. Huang, T.C. Wang, B.Y. Hsieh, and L.G. Chen: Hardware Architecture Design for Variable Block Size Motion Estimation in MPEG-4 AVC/JVT/ITU-T H.264, Proceedings of the 2003 International Symposium on Circuits and Systems, Vol. 2. (2003) 796–799
5. C.Y. Kao and Y.L. Lin: An AMBA-Compliant Motion Estimator for H.264 Advanced Video Coding, Proceedings of 2004 International Conference on SoC Design, Vol. 1. (2004) 117–120
6. C. Wei, M.Z. Gang: A Novel SAD Computing Hardware Architecture for Variable-size Block Motion Estimation and Its implementation, Proceedings of 5th International Conference on ASIC, Vol. 2. (2003) 950–953
7. Peter Kuhn: Algorithms, Complexity Analysis and VLSI Architectures for MPEG-4 Motion Estimation, Kluwer Academic Publishers, U.S.A (2003) 18–19
8. Iain E. G. Richardson: H.264 and MPEG-4 Video Compression, Wiley, UK (2004) 170–172
9. J.F Shen, T.C. Wang, and L.G. Chen: A Novel Low-Power Full-Search Block-Matching Motion-Estimation Design for H.263+, IEEE Transactions on Circuits and Systems for Video Technology, Vol. 11, No.7. (2001)

Embedded Intelligent Imaging On-Board Small Satellites

Siti Yuhaniz, Tanya Vladimirova, and Martin Sweeting

Surrey Space Centre, School of Electronics and Physical Sciences,
University of Surrey, GU2 7XH, UK
{s.yuhaniz, t.vladimirova, m.sweeting}@surrey.ac.uk

Abstract. Current commercial Earth Observation satellites have very restricted image processing capabilities on-board. They mostly operate according to a 'store-and forward' mechanism, where the images are stored on-board after being acquired from the sensors and are downlinked when contact with a ground station occurs. However, in order for disaster monitoring satellite missions to be effective, there is a need for automated and intelligent image processing on-board. In fact, the need for increasing the automation on-board is predicted as one of the main trends for future satellite missions. The main factors that hold back this concept are the limited power and computing resources on-board the spacecraft. This paper reviews existing image processing payloads of earth observing small satellites. An autonomous change detection system is proposed to demonstrate the feasibility of implementing an intelligent system on-board a small satellite. Performance results for the proposed intelligent imaging system are estimated, scaled and compared to existing hardware that are being used in the SSTL DMC satellite platform.

1 Introduction

Earth observing (EO) satellites are satellites, the main task of which is to observe the Earth by capturing images of the Earth surface using various imagers. Most satellites of this type, such as Landsat, SPOT and IKONOS are large and expensive missions, taking many years to develop. However, small satellites are emerging that are becoming a better option than the large satellites because of their lower costs, shorter development time and very good imaging sensors.

Present Earth Observation satellites are lagging behind terrestrial applications in their image processing capabilities. They mostly operate according to a 'store-and forward' mechanism, where the images are stored on-board after being acquired from the sensors and are downlinked when contact with a ground station occurs. However, in order for disaster monitoring satellite missions to be effective, there is a need for automated and intelligent image processing on-board. In fact, the need for increasing the automation on-board is predicted as one of the main trends for future satellite missions [1]. The major factors that hold back this concept are the limited power and computing resources on-board the spacecraft.

It is predicted that future satellite missions will be capable of carrying out intelligent on-board processing such as image classification, compression and change detection. The ability to detect temporal changes in images is one of the most important functions in intelligent image processing systems for hazard and disaster monitoring

T. Srikanthan et al. (Eds.): ACSAC 2005, LNCS 3740, pp. 90 – 103, 2005.

applications. Change detection analysis that requires two or more images (multispectral or SAR) acquired over time has recently being adopted in various applications. Flooded areas can be recognized using several multispectral images [2]. Change detection analysis has also been used to detect and assess earthquake's damage to an urban area [3]. City administrator can manage urban development and planning from change detection analysis results, integrated with geographic information system to monitor the urbanization growth [4]. Change detection can also help to improve the transmission bandwidth by sending to ground only the part of the image, which contains the identified changes, referred to as change image.

The Surrey Space Centre (SSC) is in close collaboration with the Surrey Space Technology Limited (SSTL), a commercial manufacturer of small satellites. SSTL is currently deploying the Disaster Monitoring Constellation (DMC), which at present consists of four micro-satellites. The DMC program offers the possibility for daily revisiting of any point on the globe. From a low Earth orbit (LEO), each DMC satellite provides 32 meter multispectral imaging, over a 600 km swath width, which is comparable with the medium resolution multispectral imagery provided by the Landsat-ETM 2, 3 and 4 missions.

This paper presents the results of a research project that aims to develop an automatic change detection system for intelligent processing of multispectral images on-board small EO satellites. In particular, we investigate the computing requirements of the imaging system of satellites based on the DMC platform. The paper is organized as follows. Section 2 reviews current trends and capabilities of on-board image processing systems on recent small satellites missions. Section 3 explains the concept of the proposed automatic change detection system for on-board intelligent imaging. Section 4 presents performance results for imaging algorithms and discusses required computing resources. Lastly conclusions are presented in section 5.

2 Image Processing On-Board Small Satellites

Intelligent imaging capabilities have already been incorporated in several Earth observing satellite missions. For example small satellites such as UoSat-5, BIRD and PROBA are carrying on-board experimental imaging payloads. This section reviews current trends and capabilities of on-board image processing systems on recent small satellites missions.

In 1991 SSTL launched the UoSAT-5 mission, which carried an advanced Earth Imaging System (EIS) consisting of a meteorological-scale Charge Coupled Device (CCD) imager and a Transputer Image Processing Experiment (TIPE) [5] [6]. The TIPE image processing unit comprises two 32-bit T800 INMOS transputers (prime processor T1 and co-processor T0), which can operate either individually or as a pair. The transputers have processing speed of 20 MHz, with performance of 30 MIPS and are equipped with 2 MBytes of CMOS SRAM. This provides a base for running powerful parallel image processing software or for implementing a backup system, in case the other transputer failed. The TIPE unit is able to control and schedule imaging operations and to perform on-board image analysis, processing and compression. Also, automatic filtering routines have been implemented. Fig. 1 shows the structure of the TIPE unit and its connection to the on-board data handling (OBDH) system.

The UoSAT-5 on-board imaging architecture is implemented on other SSTL satellite missions. For example, TiungSAT-1 [7], which was launched in 1998 had two Earth Imaging Systems - Multi-Spectral Earth Imaging System (MSEIS) and Meteorological Earth Imaging System (MEIS). TiungSAT-1 carried two transputers (T805) with 20 MHz clocking speed and 4 MBytes SRAM as the processor for the EISs and was capable of autonomous histogram analysis ensuring optimum image quality and dynamic range, image compression, autonomous cloud-editing and high compression thumb-nail image previews.

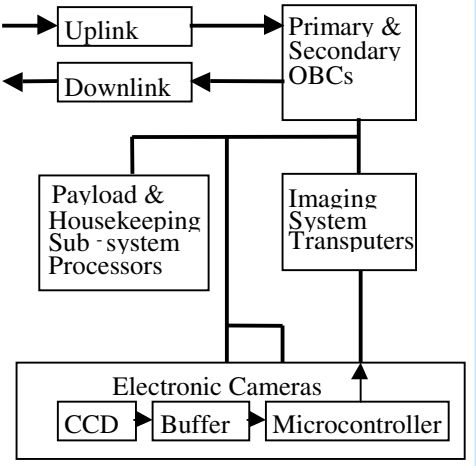


Fig. 1. Block-diagram of the on-board data handling hierarchy of UoSAT-5

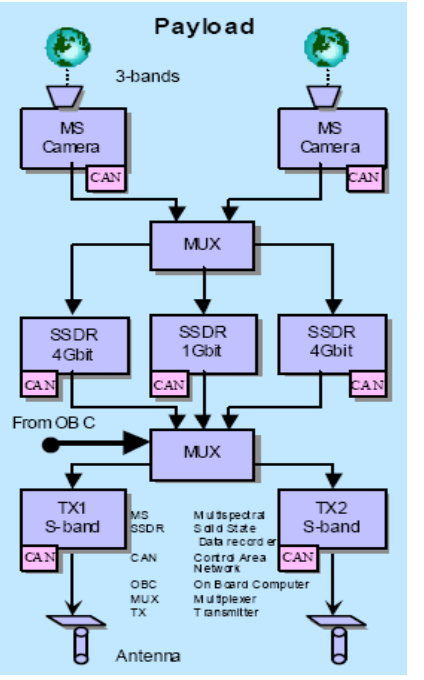


Fig. 2. Imaging data flow in the payload of a DMC-1 satellite

The SSTL's first generation of Disaster Monitoring Constellation (DMC-1) satellites used a different architecture for its imaging system. The present satellites in the DMC program are AlSat-1, BilSat-1, UKDMC-1 and NigeriaSat-1. The size of the images captured by a DMC satellite is huge, ranging from 20 Mbyte to 1 Gbyte depending on the requests from ground stations. The DMC payload manages the high image volume by splitting the images into tiles of size 2500 x 2500 pixels each. If the power is sufficient, the payload storage can accommodate 48 tiles (900 MBytes). However, on a normal imaging request, a few image tiles will be selected with up to 24 image tiles being supported during a single image taking at every orbit [8]. Significant new feature in the imaging architecture of the DMC-1 satellites is the Solid State Data Recorder (SSDR) unit.

The SSDR [9] is a general-purpose data recording device for space applications, containing a mass memory block and providing processing capabilities. It supports multiple data inputs and can store 0.5 Gbytes or 1 Gbytes of imaging data. It is designed to capture and retain raw imaging data and then downlink it on subsequent ground station passes. Two types of SSDRs are included to provide redundancy and to introduce alternate technology on-board – one is based on the Motorola PowerPC MPC 8260 and the other one is based on the Intel StrongArm SA1100 processor. Fig. 2 shows the imaging data flow in the payload of a DMC-1 satellite.

The BIRD satellite that was developed by the German Space Agency is another small satellite with image processing on-board. The imaging system of the BIRD satellite is based on two Infrared sensors and one CCD camera. Distinctive feature is the specialised hardware unit based on the neural network processor NI1000, which was integrated in the Payload Data Handling System (PDH) of the satellite. The PDH is a dedicated computer system responsible for the high-level command distribution and the science data collection between all payloads at the BIRD satellite. The neural network processor implements an image classification system, which can detect fire and hotspots.

The recent small satellite mission of the European Space Agency (ESA) Proba [10] has advanced autonomy experiments on-board. The Compact High Resolution Imaging Spectrometer (CHRIS), an EO instrument, demonstrated on-board autonomy in terms of Attitude and Orbit Control System (AOCS) data handling and resource management. The images from CHRIS are processed in a DSP based Payload Processing Unit (PPU) operating at 20 MHz. The PPU provides 1.28 Gbit (164 MBytes) of mass memory and acts as the main processing block for all on-board cameras and other payload sensors.

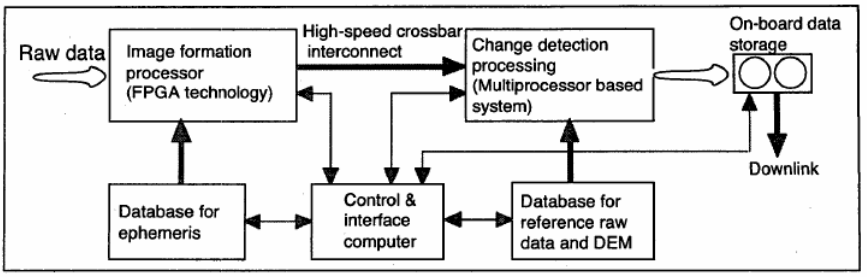


Fig. 3. Functional block diagram of the on-board change detection processors for TechSat21 [11]

A significant milestone with respect to on-board intelligent processing is the NASA autonomous Sciencecraft concept, which includes the development of the Techsat21 satellite mission. The project also includes the development of on-board science algorithms such as image classification, compression, and change detection. Under this program a specialized processor for change detection is being developed [11] which is implemented on a multiprocessor system. The hardware is based on a hybrid architecture that combines FPGAs and distributed multiprocessors. Customised FPGA cards and high-speed multiprocessors are being developed because of the limited on-board memory (512 Mbytes or less) and the relatively slow process-

ing speed of the current commercial-off-the-shelf (COTS) components. For the change detection task, the required processor performance is 2.4 GFLOPS and the required memory capacity is 4.5 Gbytes. It is aimed that the multiprocessor card will have 4 to 8 Gbytes on-board memory for the change detection processing task. Fig. 3 illustrates the functional block diagram of the proposed change detection processors.

Another development in on-board intelligent processing is the Parallel Processing Unit (PPU) of the small satellite X-Sat, which is being developed by the Nanyang Technological University in Singapore. The Parallel Processing Unit is one of the payloads of X-Sat, which is aimed at high precision Attitude Determination and Control operations, high speed communications and real-time imaging processing applications. The processing unit comprises twenty SA1110 StrongArm processors that are interlinked by FPGAs and clocked at 266 MHz and 64 Mbyte of SDRAM memory [12].

FedSat is an Australian scientific small satellite that carries High Performance Computing payload (HPC-1). In this mission, a cloud detection system [13] and a lossless image compression [14] was implemented using reconfigurable computing. The cloud detection system uses the Landsat 7 Automatic Cloud Cover Assessment (ACCA) algorithm. The cloud detection and lossless image compression algorithms were implemented in Matlab, and synthesized for the Xilinx XQR4062 FPGA in the HPC-1.

Nowadays the image processing technology in terrestrial applications has reached a very advanced stage and has been utilized in different real-time embedded applications. However, the technology used on-board spacecraft is generally less advanced compared to terrestrial applications. The advent of small satellites has made it possible to make use of new technology but with a lower risk than conventional satellites. There are a few potential technological areas, which have gained recognition in terrestrial applications that might be useful to implement on-board a satellite and intelligent imaging is one of them. Table 1 illustrates the trend in image processing on-board small satellites in terms of functionality. It can be seen that more on-board image processing functions are included in future missions. This is made possible due to availability of more powerful computing resources on-board as shown in Table 2.

Table 1. Functionality trend in on-board image processing on small satellites

Satellite	Launched Year	Image Compression	Change Detection	Recognition and Classification
UoSat-5	1991	Yes	No	No
BIRD	2000	No	No	Yes
PROBA	2001	Yes	No	Yes
FEDSAT	2002	Yes	No	Yes
UK-DMC	2003	Yes	No	No
X-SAT	2006	Yes	Yes	Yes
Techsat21	2008	Yes	Yes	Yes

Table 2. Computing characteristics of image processor payloads on-board small satellites

Satellite	Processors Speed	Main Memory (MBytes)	Parallelism
UoSat-5	20 MHz	4	No
BIRD	33 MHz	8	No
PROBA	20 MHz	128	No
FEDSAT	20 MHz	1	No
UK-DMC	100 MHz	1	No
X-SAT	266 MHz	64	Yes
Techsat21	133 MHz	128	No

3 Automatic Change Detection System for On-Board Imaging

In order to investigate and demonstrate the feasibility of on-board intelligent imaging, we are developing an automatic change detection system (ACDS). This system will provide capability for detecting temporal changes in newly captured images based on comparison with reference images, which will be retrieved from an on-board database. Such a capability is extremely useful for disaster monitoring and warning applications and can form the nucleus of such on-board systems, however there are other advantages to it. Normally it is the image compression capability that can serve the purpose of increasing the transmission bandwidth. The availability of an automatic change detection system on-board a satellite can also help in that as only the change images can be sent to ground instead of the whole images. Other possible advantage of having change detection on-board is using the changed/unchanged information in conjunction with the image scheduling system. This can allow the satellite to dynamically reschedule the scanning of an area if the recently captured image has been recognised as changed and represents an area of interest.

Fig. 4 shows the flow chart of the proposed change detection system. The rectangular blocks denote the main processing tasks and the parallelogram blocks denote the processing data. A new image from the satellite cameras will go through several processing steps: image tiling, registration and change detection to generate changed and unchanged tiles of the image The system also includes a database for keeping previously taken images for reference purposes.

There are two ways, in which the images can enter the ACDS - either directly from the imagers, or from temporary storage. The system contains three main processing blocks - image tiling, pre-processing and change detection. When new images enter ACDS, they are split into tiles of size 500 x 500 pixels. In the pre-processing block, the images are enhanced and co-registered so that they are suitable for processing in the change detection subsystem. Co-registration is the process of aligning the newly acquired image with a base image, which is a previously acquired image retrieved from the on-board database.

For the sake of simplicity, we assume that the image registration operation is able to detect rigid body misalignment only, which is a result of translation, rotation and

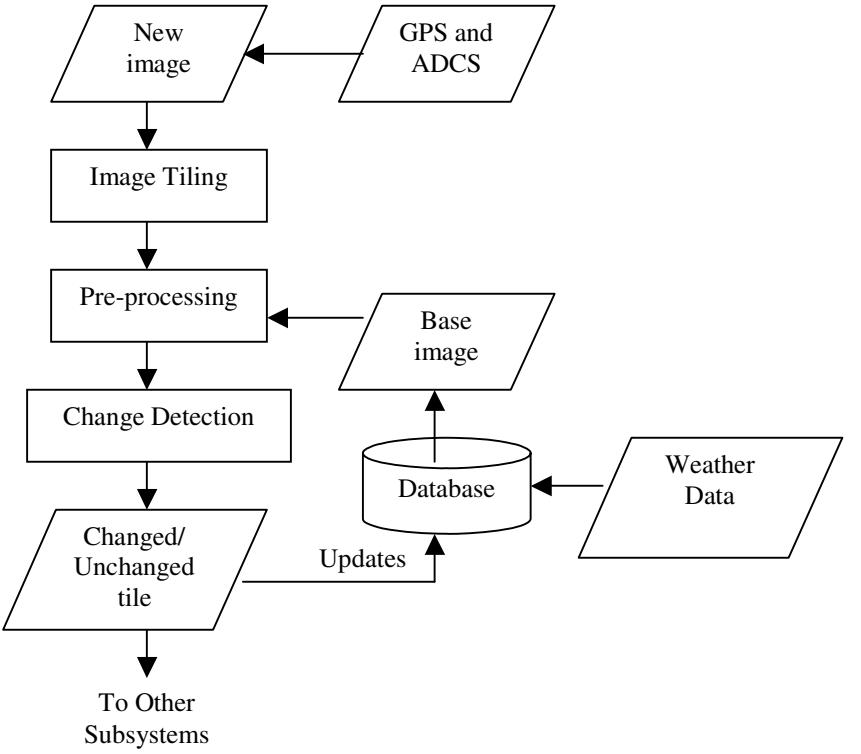


Fig. 4. Flow chart of the on-board automatic change detection system (ACDS)

scaling transformations. Five image registration algorithms have been selected as the candidate methods for the image registration block for their effectiveness in correcting rigid body miosalignment:

1. *Cross correlation*. A measure of similarity between pixels on input image and base image to find the translation shift. It is a slow method and sensitive to noise.
2. *Phase correlation.* Phase correlation works by computing the Fourier transform for the input and base image and finding the peaks of their inverse cross power spectrum [15]. It is a fast method, however, it is only useful for registration of images that have translation misalignment.
3. *Mutual information*. This method is based on probability theory and information theory. It is a measure of statistical dependency between two data sets and is very useful for registration of multi-modal images [16].
4. *Fourier-Mellin registration.* This is an extension of the phase correlation algorithm adding the capability to detect rotation and scaling misalignment between input and base images [17].
5. *Linear point mapping*. This registration method is based on using control points selected on the input and base images. The control points will be used to generate the linear polynomial that will match input image to base image [18].

The pre-processed image will go to the change detection processing block, which produces the change image. The change image will be saved in the database and will be used as the reference image for the next processing task. Several change detection algorithms have been selected as candidate methods for this processing block among which image differencing has produced the most optimal results [19].

1. *Image differencing*. This method subtracts each pixel value in one image from the corresponding pixel value in the second image to get the change image.
2. *Image rationing*. The change image is generated by dividing one image by the other.
3. *Image regression.* The input image is linearly regressed to the base image and the change image pixel values are obtained from the residual of the regression. Difference in atmospheric condition and illumination can be reduced by using this method.
4. *Change vector analysis.* A multivariate change detection technique, which processes spectral and temporal aspects of the image data.
5. *Principal component analysis*. A technique that allows the production of images where the correlation between them is zero [20]. The change information is usually on the third and later principal components.

We are also introducing a database in the ACDS to supply the base image and also for keeping weather data, which are necessary for particular remote sensing applications. On-board databases is a relatively new technology for spacecraft engineering, especially in small satellites development because hard discs contains moving parts and are very fragile to be flown on spacecraft. However, there are several research projects and attempts to put high volume mass memory in space. For instance, IMT [21] is developing a 73 Gbyte hard disc for spacecraft applications. This volume would be sufficient to implement a database that contains captured images and weather data. SSTL are flying hard disc drives on the China DMC satellite, which is to be launched in September 2005, to test the technology.

Weather data are planned to be used in future missions where the satellite can receive data from various sources such as ground stations and perhaps other satellites using intersatellite links. The weather data can be useful in predicting cloud covers and natural disaster such as flooding, combined with the change detection results.

The automatic change detection system is aimed at satisfying the following requirements:

1. The change detection methods have to be fully automated, as no direct human intervention could be provided on-board.
2. The system should be fast enough to register and detect changes using a pair of image with the size of 20,000 x 15,000 pixels, which is the maximal size of an SSTL raw DMC image. The output of the automatic change detection system will be downloaded when the satellite is within view of the ground station. For the purpose of estimating the required processing time we will make the following assumptions:

- Although the relative position of the ground station and the target area varies, we will assume that the ground station is in the centre of the target area.
- The shortest time for the satellite to reach back the ground station is one orbit, which is about 90 minutes for LEO.
- The typical contact time between a LEO satellite and the ground station is 12 minutes.

So, the system should be able to process the raw image into change detection output within 78 minutes (90 minus 12 minutes) before it reaches the ground station.

3. The methods are feasible to be implemented on the China DMC and the Alsat-2 satellite platforms.

On the DMC platform the software implementation of the system can be executed in the SSDR units. The PowerPC based SSDR unit will be used for the performance evaluation presented in the following section. The PowerPC processor is capable of 280 Dhrystone MIPS at 200 MHz and has 1 Mbyte of RAM.

4 Performance Evaluation

This section presents performance results for the investigated registration and change detection algorithms and discusses the required computing resources on-board.

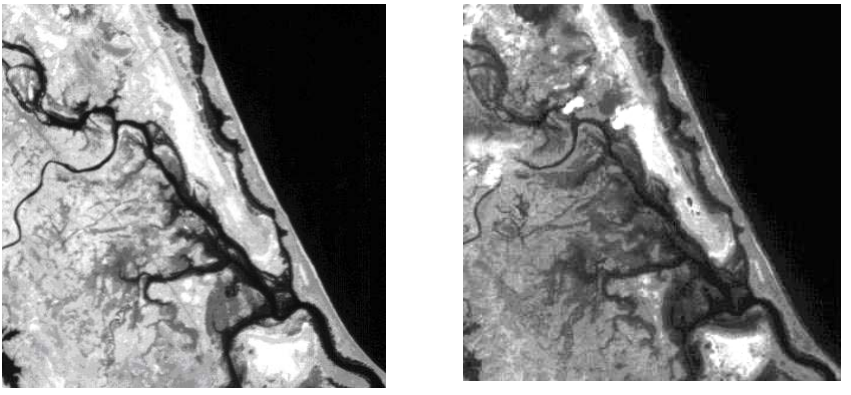


Fig. 5. Near Infrared images of Quang Nai, Vietnam in July (left) and in October (right) 2003 captured by the UK-DMC satellite

At the current stage of the work, the pre-processing and change detection subsystems of the proposed systems are being investigated. The selected numerical methods for change detection and image registration, as detailed in section 3 above, were implemented using Matlab and were compared to each other to find the required processing time and amount of memory. The software was executed on a Pentium M 1.3

GHz personal computer since flight test hardware was not available to use. In order to estimate the performance of the SSDR PowerPC the Dhrystone 2.1 benchmark program was run on the test Pentium computer resulting in 1665 Dhrystone MIPS. The experimental results were then scaled down to reflect the DMC SSDR performance, using the MIPS rate of the SSDR PowerPC processor.

The test images comprise a pair of SSTL DMC images of Quang Nai, Vietnam in three bands (Near Infra Red, Red and Green) of size 500 x 500 pixels each. As shown in Fig 5 the test images depict the before and after flooding event in July and October 2003.

Table 3 and Table 4 show the performance results for the registration and change detection numerical experiments respectively. In order to estimate the amount of the required memory, the memory is divided into two categories – image memory and data memory. The image memory is used to store the input images and the output images, while the variables memory is used to store the variables and constants for the calculations that are employed by the algorithms. The memory to store the program itself is not taken into account in this study.

The processing time for the registration methods is the time needed for each method to estimate how much the new image has been misaligned with respect to the reference image, and also the time it takes to transform the image into a registered image. In Table 3 the first three algorithms - phase correlation, cross correlation and mutual information - are used to find translation misalignment, while the other two algorithms - Fourier-Mellin and linear point mapping are used to find translation, rotation and scaling misalignments. The control points in the linear point mapping method were selected manually and therefore the processing time for this method in Table 3 does not account for the duration of the control points' selection process, which makes the result too optimistic. Automated solutions to the control point selection problem are needed since manually selecting control points cannot be done onboard. Phase correlation works fast, but requires more memory. Mutual information is slow, but less memory is required. The estimation of the processing time for the mutual information method is quite tricky because it depends on how we set the parameters. The processing time for the mutual information method in Table 4 is the time when 200 pairs of pixels from the new and the reference image are selected to be computed. All five registration methods give similar accuracy results, although it has to be noted that the Fourier-Mellin and the linear point mapping methods recover all rigid body types of misalignment.

Three of the five change detection algorithms in Table 4 - image differencing, image rationing and image regression – require processing of only one spectral band, while the other two tested change detection algorithms require all three bands. The processing time was measured from the time the input images have been read into the algorithm to the time of producing the change images. As shown in Table 4, image differencing is the fastest method, along with image rationing. In terms of accuracy, image differencing gives the lowest false alarm and lowest overall error [19].

The processing times in Tables 3 and 4 are measured based on execution of Matlab code and therefore they might not be as accurate as embedded software implementations using C or Assembly languages. In particular, the execution of the principal component analysis will take longer on the target CPU since the Matlab model uses built-in functions to find the covariance matrix, eigenvalues and eigenvectors.

Table 3. Performance of image registration methods using 500 x 500 pixel images

Algorithm	Estimated Memory (MBytes)			Time at 1665 MIPS (sec)	Time at 280 MIPS (sec)
	Image Data	Vari-ables	Total		
Phase correlation	0.75	16.00	16.75	0.70	4.16
Cross correlation	0.75	8.00	8.75	5.80	34.49
Mutual information	0.75	9.25	10.00	30.40	178.39
Fourier-Mellin registration	0.75	21.00	21.75	4.92	29.26
Linear point mapping	0.75	0.00	0.75	1.52	9.04

Table 4. Performance of change detection methods using 500 x 500 pixel images

Algorithm	Estimated (MBytes)		Memory	Time at 1665 MIPS (sec)	Time at 280 MIPS (sec)
	Image Data	Vari-ables	Total		
Image differencing	0.75	0.00	0.75	0.08	0.48
Image rationing	0.75	0.00	0.75	0.08	0.48
Image regression	0.75	3.84	4.59	0.50	2.97
Change vector analysis	1.75	0.00	1.00	32.00	190.29
Principal component analysis	1.75	0.30	2.05	1.80	10.70

A DMC image of maximal size comprises 1,200 tiles of 500 x 500 pixels. If we assume that we have to detect all the rigid body misalignments (translation, rotation and scaling), the Fourier-Mellin registration would be selected as the optimal method. The linear point mapping method would actually take longer than indicated in Table 3 if the control point selection process is considered. The image differencing is selected as the change detection method because it is one of the fastest methods as shown in Table 4 and also it achieves a good accuracy [19]. If the image differencing and the Fourier-Mellin registration algorithms are used the total processing time for a DMC image of maximal size is 9.9 hours, as follows:

$$(29.26 \text{ sec} + 0.48 \text{ sec}) \times 1{,}200 \text{ tiles} = 35{,}688 \text{ sec} = 9.9 \text{ hours} \quad (1)$$

The obtained estimate of 9.9 hours for the processing time of the two subsystems is significantly higher than the targeted processing time of 78 minutes as detailed in section 3 above. A faster solution is required, which could be achieved with a high performance parallel computing architecture. Such an architecture should also be low-power because of the limited power budget on-board a satellite.

Fig. 6 shows a diagram of a possible multiprocessor parallel architecture based on the flight-proven PowerPC processor. It consists of fifteen MPC8260 processors

connected in parallel, whereby each processor has 1M of RAM, is capable of 280 MIPS at 200 MHz and operates at a low voltage of 2.5V. This concept is based on the current DMC SSDR specification, but the processing is undertaken in a parallel way to introduce faster processing. Some of the advantages of the proposed architecture are listed below:

i) The image is split into several tiles and each processor handles one image tile at a time. If all fifteen processors are used, the processing time for a maximal size image will be approximately 40 minutes, which is about half of the required execution time.

ii) Certain processors can be turned off when the input images are not so huge in size, which depends on the satellite mission.

This architecture might consume more power than a single-processor architecture and therefore it will require the incorporation of a power management mechanism. The system discussed here is a generic system suitable for any disaster monitoring application. It can include other autonomous image processing tasks such as classification, compression, encryption, etc which will have to be optimized to fit in the remaining processing time. The specialized image processing will depend on the nature of the monitoring event, for example flood detection.

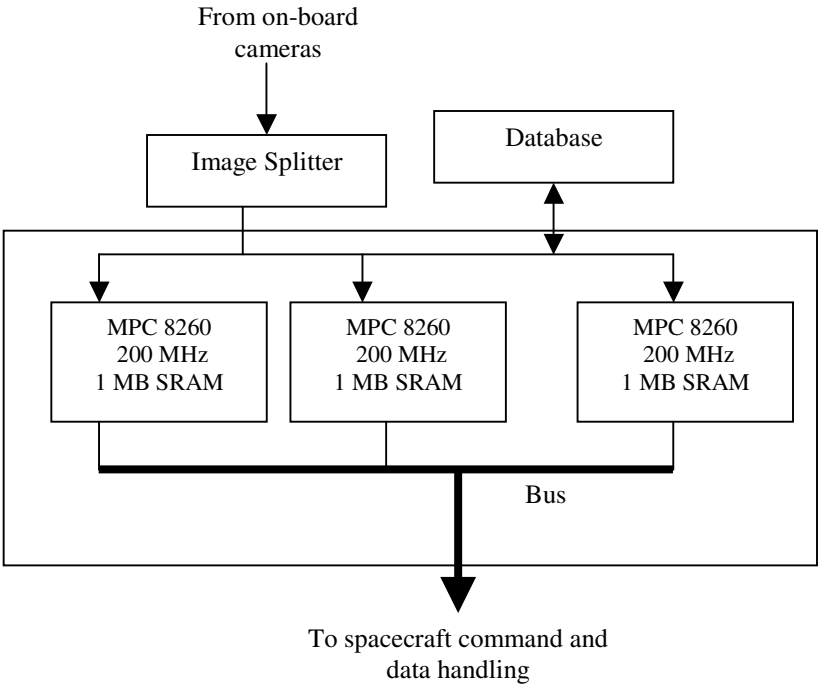


Fig. 6. Parallel computing architecture for intelligent imaging on-board small satellites

5 Conclusions

The future trend in on-board imaging systems of space missions is to achieve partial or complete autonomy. Although the imaging capabilities of present spacecraft are still limited, experimental small satellites have been flown in orbit in order to test new hardware and software. There are many research projects and experimental developments, which are aimed at transferring terrestrial technology to on-board the spacecraft. Although the limitations of small satellites in terms of mass, volume and power make the implementation of intelligent imaging capability on-board a difficult task, the advent of new electronics technology and lower-cost small satellites may make the impossible become possible in the near future.

This paper presented an automatic change detection system for on-board implementation and the results of a feasibility study into intelligent imaging on-board a small satellite. Processing time for two main processing tasks in the system was measured and scaled to the computing capabilities of the current SSTL DMC satellite platform. The next phase of the project will be focusing on developing a decision making subsystem, which will consist of flood detection and image compression blocks.

References

1. Zhou, G., and Kafatos, M.: Future Intelligent Earth Observing Satellites. Proceedings of ISPRS Commission 1 FIEOS Conference XXXIV(1) (2002)
2. Yoon, G-W.; Yun, Y-B., And Park, J-H.: Change Vector Analysis: Detecting Of Areas Associated With Flood Using Landsat TM. Proceedings of 2003 IEEE International Geoscience and Remote Sensing Symposium, IGARSS'03, Vol. 5. (2003) 3386-3388
3. Zhang, J-F., Xie, L, And Tao, X.: Change Detection Of Earthquake-Damaged Buildings On Remote Sensing Image And Its Application In Seismic Disaster Assessment. Proceedings of 2003 IEEE International Geoscience and Remote Sensing Symposium, IGARSS'03. Vol. 4. (2003) 2436-2438
4. Gonzalez, J, Ambrosio, I., And Arevalo, V.: Automatic Urban Change Detection From The IRS-1D PAN. IEEE/ISPRS Joint Workshop 2001 Remote Sensing and Data Fusion over Urban Areas (2001) 320-323
5. Fouquet, M.: The UoSAT-5 Earth Imaging System – In Orbit Results SPIE Proc. Conference on Small Satellite Technology and Applications, Orlando, Florida, USA (1992)
6. Montpetit, M., Prieur, L., Fouquet, M.: Compression of UoSAT-5 Images. SPIE Proc. Conference on Small Satellite Technology and Applications, Orlando, Florida, USA (1993)
7. Othman, M. and Arshad, A: TiungSAT-1: From Inception to Inauguration Astronautic Technology (M) Sdn Bhd, Malaysia (2001)
8. da Silva Curiel, A., Boland, L., Cooksley, J., Bekhti, M., Stephens, P., Sun, W. and Martin, S.: First Results From The Disaster Monitoring Constellation (DMC). 4th IAA Symposium on Small Satellites for Earth Observation, Berlin, Germany (2003)
9. Solid State Data Recorder Datasheets, http://www.sstl.co.uk/documents/MPC8260 Solid State Data Recorder.pdf
10. Bermyn, J.: PROBA - project for on-board autonomy. Air & Space Europe. Vol. 2, Iss. 1, (2000) 70-76

11. Lou, Y., Hensley, S., Le, C., Moller, D: On-board processor for direct distribution of change detection data products [radar imaging]. Proceedings of the IEEE Radar Conference (2004) 33- 37
12. Bretschneider, T.: Singapore's Satellite Mission X-Sat. 4th IAA Symposium on Small Satellites for Earth Observation, Berlin, Germany (2003)
13. Williams, J.A. Dawood, A.S. Visser, S.J: FPGA-based cloud detection for real-time on-board remote sensing. Proceedings of 2002 IEEE International Conference on Field-Programmable Technology (FPT). (2002) 110-116
14. Dawood, A.S., Williams, J.A., Visser, S.J.: On-board satellite image compression using reconfigurable FPGAs. Proceedings of 2002 IEEE International Conference on Field-Programmable Technology (FPT). (2002) Pages: 306-310
15. Kuglin, C. D., and Hines, D. C.: The phase correlation image alignment method. Proceedings of the IEEE 1975 International Conference on Cybernetics and Society. IEEE, New York (1975) 163-165
16. Zitova, B. and Flusser, J.: Image registration methods: a survey. Image and Vision Computing, Vol. 21, (2003) 977-1000
17. Reddy, B.S.; Chatterji, B.N. 1996. An FFT-based technique for translation, rotation, and scale-invariant image registration. IEEE Transactions on Image Processing. Vol. 5, Iss. 8. IEEE New York (1996) 1266-1271
18. Brown, L.G.: A survey of image registration techniques. ACM Computing Surveys. Vol. 24 (1992) 326-376
19. Yuhaniz, S., Vladimirova T., Sweeting M.N.: Automatic Change Detection for Multispectral Images. Proceedings of National Conference on Computer Graphics and Multimedia COGRAMM 2004, Kuala Lumpur (2004) 52-58
20. Gibson, P., and Power, C.: Introductory Remote Sensing: Digital Image Processing and Application. Routledge, London (2000)
21. Perrota, G. and Cucinella, G.: COTS-based activities on Spacecraft Data Systems by IMT. Workshop on Spacecraft Data Systems, European Space Agency, Netherlands (2003)

Architectural Enhancements for Color Image and Video Processing on Embedded Systems*

Jongmyon Kim[1], D. Scott Wills[2], and Linda M. Wills[2]

[1] Chip Solution Center, Samsung Advanced Institute of Technology,
San 14-1, Nongseo-ri, Kiheung-eup, Kyungki-do, 449-712, South Korea
jongmyon.kim@samsung.com
[2] School of Electrical and Computer Engineering,
Georgia Institute of Technology, Atlanta, Georgia 30332-0250
{scott.wills, linda.wills}@ece.gatech.edu

Abstract. Application-specific extensions of a processor provide an efficient mechanism to meet the growing performance demands of multimedia applications. This paper presents a color-aware instruction set extension (CAX) for embedded multimedia systems that supports vector processing of color image sequences. CAX supports parallel operations on two-packed 16-bit (6:5:5) YCbCr (luminance-chrominance) data in a 32-bit datapath processor, providing greater concurrency and efficiency for color image and video processing. Unlike typical multimedia extensions (e.g., MMX, VIS, and MDMX), CAX harnesses parallelism within the human perceptual YCbCr space, rather than depending solely on generic subword parallelism. Experimental results on an identically configured, dynamically scheduled 4-way superscalar processor indicate that CAX outperforms MDMX (a representative MIPS multimedia extension) in terms of speedup (3.9× with CAX, but only 2.1× with MDMX over the baseline performance) and energy reduction (68% to 83% reduction with CAX, but only 39% to 69% reduction with MDMX over the baseline). More exhaustive simulations are conducted to provide an in-depth analysis of CAX on machines with varying issue widths, ranging from 1 to 16 instructions per cycle. The impact of the CAX plus loop unrolling is also presented.

1 Introduction

With the proliferation of color output and recording devices (e.g., digital cameras, scanners, and monitors) and color images on the World Wide Web (WWW), a user can easily record an image, display it on a monitor, and send it to another person over the Internet. However, the original image, the image on the monitor, and the image received through the Internet usually do not match because of faulty display or channel transmission errors. Color image processing methods offer solutions to many of the problems that occur in recoding, transmitting, and creating color images. However, these applications demand tremendous computational and I/O throughput. Thus, understanding the characteristics of the color imaging application domain provides new opportunities to define an efficient architecture for embedded multimedia systems.

* This work was performed by author at the Georgia Institute of Technology (Atlanta, GA).

T. Srikanthan et al. (Eds.): ACSAC 2005, LNCS 3740, pp. 104 – 117, 2005.

Manufactures of general-purpose processors (GPPs) have included multimedia extensions (e.g., MMX [12], VIS [16], ALTIVEC [11], and MDMX [9]) to their instruction set architectures (ISAs) to support multimedia applications. These multimedia extensions exploit subword parallelism by packing several small data elements (e.g., 8-bit pixels) into a single wide register (e.g., 32-, 64-, and 128-bit), while processing these separate elements in parallel within the context of dynamically scheduled superscalar instruction-level parallel (ILP) machine. However, their performance is limited in dealing both with color data that are not aligned on boundaries that are powers of two (e.g., adjacent pixels from each band are visually spaced three bytes apart) and with storage data types that are inappropriate for computation (necessitating conversion overhead before and usually following the computation) [13]. Although the band separated format (e.g., the red data for adjacent pixels are adjacent in memory) is the most convenient for single instruction, multiple data (SIMD) processing, a significant amount of overhead for data alignment is expected prior to SIMD processing. Even if the SIMD multimedia extensions store the pixel information in the band-interleaved format (i.e., |Unused|R|G|B| in a 32-bit wide register), subword parallelism cannot be exploited on the operand of the unused field, shown in Figure 1(b). Furthermore, since the RGB color space does not model the perceptual attributes of human vision well, the RGB to YCbCr conversion is required prior to color image processing [4]. Although the SIMD multimedia extensions can handle the color conversion process in software, the hardware approach would be more efficient.

This paper presents a novel color-aware instruction set extension (CAX) for embedded multimedia systems to solve problems inherent to packed RGB extensions by supporting parallel operations on two-packed 16-bit (6:5:5) YCbCr (luminance-chrominance) data in a 32-bit datapath processor, resulting in greater concurrency and efficiency for color imaging applications. The YCbCr space allows coding schemes that exploit the properties of human vision by truncating some of the less important data in every color pixel and allocating fewer bits to the high-frequency chrominance components that are perceptually less significant. Thus, the compact 16-bit color representation consisting of a 6-bit luminance (Y) and two 5-bit chrominance (Cb and Cr) components provides satisfactory image quality [6, 7].

This paper evaluates CAX in comparison to a representative multimedia extension, MDMX, an extension of MIPS. MDMX was chosen as a basis of comparison because it provides an effective way of dealing with reduction operations, using a wide packed accumulator that successively accumulates the results produced by operations on multimedia vector registers. Other multimedia extensions (e.g., MMX and VIS) provide more limited support of vector processing in a 32-bit datapath processor without accumulators. To handle vector processing on a 64-bit or 128-bit datapath, they require frequent packing/unpacking of operand data, deteriorating their performance.

Experimental results for a set of color imaging applications on a 4-way superscalar ILP processor indicate that CAX achieves a speedup ranging from 3× to 5.8× (an average of 3.9×) over the baseline performance without subword parallelism. This is in contrast to MDMX, which achieves a speedup ranging from 1.6× to 3.2× (an average of 2.1×) over the baseline. CAX also outperforms MDMX in energy reduction (68% to 83% reduction with CAX, but only 39% to 69% with MDMX over the baseline version). Furthermore, CAX exhibits higher relative performance for low-issue

rates. These results demonstrate that CAX is an ideal candidate for embedded multimedia systems in which high issue rates and out-of-order execution are too expensive.

Performance improved by CAX is further enhanced through loop unrolling (LU) [3], an optimization technique that reorganizes and reschedules the loop body, which contains the most critical code segments for imaging applications. In particular, LU reduces loop overhead while exposing ILP within the loops for machines with multiple functional units. Experimental results indicate that LU (by a factor of 3 for three programs and 4 for two programs) provides an additional 4%, 19%, and 21% performance improvement for the baseline, MDMX, and CAX versions, respectively. This result suggests that the CAX plus LU technique has the potential to provide the much higher levels of performance required by emerging color imaging applications.

The rest of this paper is organized as follows. Section 2 summarizes the CAX instruction set for superscalar ILP processors. Section 3 describes our workloads, the modeled architectures, and a simulation infrastructure for the evaluation of CAX. Section 4 presents the experimental results and their analysis, and Section 5 concludes this paper.

2 A Color-Aware Instruction Set for Color Imaging Applications

The color-aware instruction set extension (CAX), applied to current microprocessor ISAs, is targeted at accelerating color image and video processing. CAX supports parallel operations on two-packed 16-bit (6:5:5) YCbCr data in a 32-bit datapath processor, providing greater concurrency and efficiency for vector processing of color image sequences. In addition, CAX employs color-packed accumulators that provide a solution to overflow and other issues caused by packing data as tightly as possible by implicit width promotion and adequate space. Figure 1 illustrates three types of operations: (1) a baseline 32-bit operation, (2) a 4×8-bit SIMD operation used in many general-purpose processors, and (3) a 2×16-bit CAX operation employing heterogeneous (non-uniform) subword parallelism. CAX instructions are classified into four different groups: (1) parallel arithmetic and logical instructions, (2) parallel compare instructions, (3) permute instructions, and (4) special-purpose instructions.

2.1 Parallel Arithmetic and Logical Instructions

Parallel arithmetic and logical instructions include packed versions of addition (`ADD_CRCBY`), subtraction (`SUBTRACT_CRCBY`), and average (`AVERAGE_CRCBY`). The addition and subtraction instructions include a saturation operation that clamps the output result to the largest or smallest value for the given data type when an overflow occurs. Saturating arithmetic is particularly useful in pixel-related operations, for example, to prevent a black pixel from becoming white if an overflow occurs. The parallel average instruction, which is useful for blending algorithms, takes two packed data types as input, adds corresponding data quantities, and divides each result by two while placing the result in the corresponding data location. The rounding is performed to ensure precision over repeated average instructions.

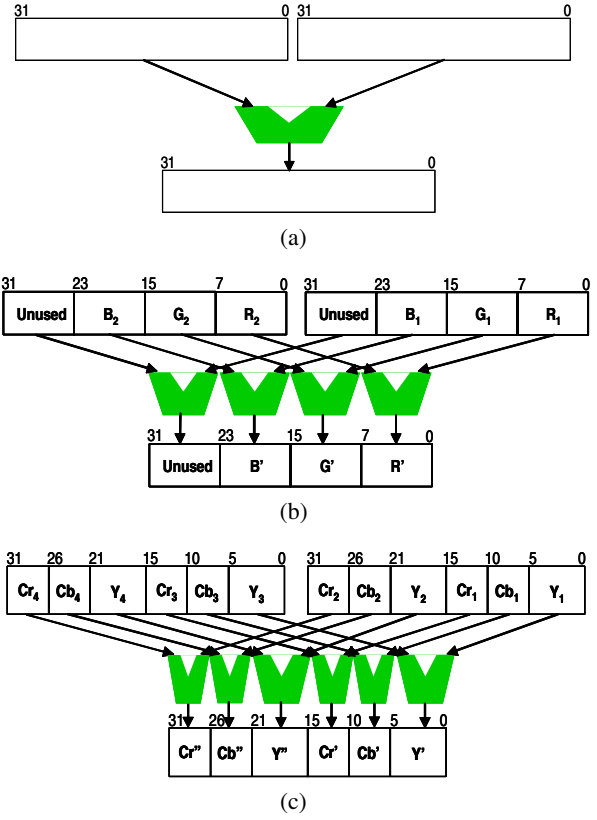


Fig. 1. Types of operations: (a) a baseline 32-bit operation, (b) a 32-bit SIMD operation, and (c) a 32-bit CAX operation

2.2 Parallel Compare Instructions

Parallel compare instructions include `CMPEQ_CRCBY`, `CMPNE_CRCBY`, `CMPGE_CRCBY`, `CMPGT_CRCBY`, `CMPLE_CRCBY`, `CMPLT_CRCBY`, `CMOV_CRCBY` (conditional move), `MIN_CRCBY`, and `MAX_CRCBY`. These instructions compare pairs of sub-elements (e.g., Y, Cb, and Cr) in the two source registers. Depending on the instructions, the results are varied for each sub-element comparison. The `CMPEQ_CRCBY` instruction, for example, compares pairs of sub-elements in the two source registers while writing a bit string of 1s for true comparison results and 0s for false comparison results to the target register. The first seven instructions are useful for a condition query performed on the incoming data such as chroma-keying [10]. The last two instructions, `MIN_CRCBY` and `MAX_CRCBY`, are especially useful for median filtering, which compare pairs of sub-elements in the two source registers while outputting the minimum and maximum values to the target register.

2.3 Parallel Permute Instructions

Permute instructions include `MIX_CRCBY` and `ROTATE_CRCBY`. These instructions are used to rearrange the order of quantities in the packed data type. The mix instruction mixes the sub-elements of the two source registers into the operands of the target register, and the rotate instruction rotates the sub-elements to the right by an immediate value. Figures 2(a) and (b) illustrate the rotate and mix instructions, respectively, which are useful for performing a vector pixel transposition or a matrix transposition [14].

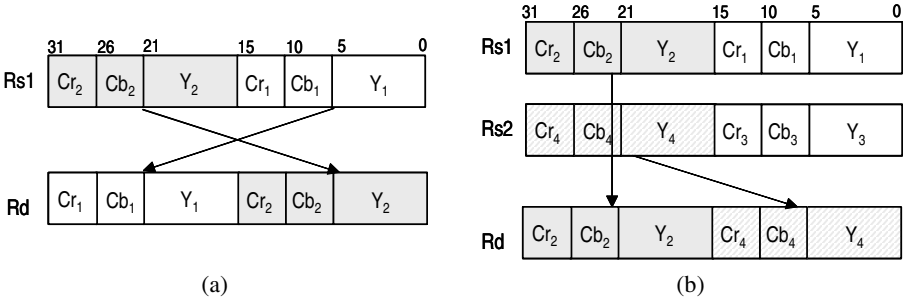


Fig. 2. (a) A rotate instruction. (b) A mix instruction

2.4 Special-Purpose Instructions

Special-purpose CAX instructions include `ADACC_CRCBY` (absolute-differences-accumulate), `MACC_CRCBY` (multiply-accumulate), `RAC` (read accumulator), and `ZACC` (zero accumulator), which provide the most computational benefits of all the CAX instructions. The `ADACC_CRCBY` instruction, for example, is frequently used in a number of algorithms for motion estimation. It calculates the absolute differences of pairs of sub-elements in the two source registers while accumulating each result in the packed accumulator, shown in Figure 3. The `MACC_CRCBY` instruction is useful in DSP algorithms that involve computing a vector dot-product, such as digital filters and convolutions. The last two instructions `RAC` and `ZACC` are related to the managing of the CAX accumulator.

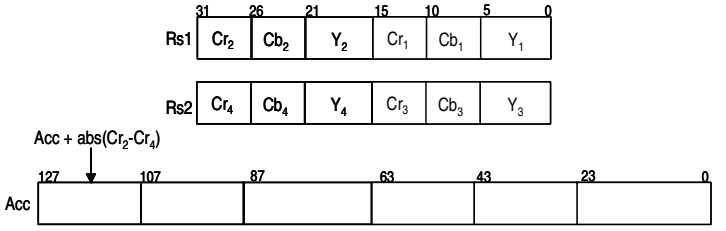


Fig. 3. An absolute-differences-accumulate instruction

3 Methodology

3.1 Color Imaging Applications

Five imaging applications, briefly summarized in Table 1, have been selected to capture a range of color imaging in multimedia: color edge detection using a vector Sobel operator (VSobel), the scalar median filter (SMF), the vector median filter (VMF), vector quantization (VQ), and the full-search vector BMA (FSVBMA) of motion estimation within the MPEG standard. (See [6] for more details.) Although the SMF is not an example of vector processing, it was included in the application suite because of its useful and well-known sorting algorithm. These applications form significant components of many current and future real-world workloads such as streaming video across the internet, real-time video enhancement and analysis, and scene-visualization. All the applications are executed with 3-band CIF resolution (352×288 pixels) input image sequences.

Table 1. Summary of color imaging applications used in this study

Benchmark	Description
VSobel	Extracts color edge information from an image through a Sobel operator that accounts for local changes in both luminance and chrominance components.
SMF	Removes impulse noise from an image by replacing each color component with a median value in a 3×3 window that is stepped across the entire image. The three resulting images are then combined to produce a final output image.
VMF	Suppresses impulse noise from an image through a vector approach that is performed simultaneously on three color components (i.e., Y, Cb, and Cr).
VQ	Compresses and quantizes collections of input data by mapping k-dimensional vectors in vector space $\mathbf{R}^k$ onto a finite set of vectors. A full search vector quantization using both luminance and chrominance components is used to find the best match in terms of the chosen cost function.
FSVBMA	Removes temporal redundancies between video frames in MPEG/H.26L video applications. A full search block-matching algorithm using both luminance and chrominance components is used to find one motion vector for all components.

3.2 Modeled Architectures and Tools

Figure 4 shows a methodology framework for this study. The Simplescalar-based toolset [1], an infrastructure for out-of-order superscalar modeling, has been used to simulate a superscalar processor without and with MDMX or CAX. To generate the MDMX and CAX versions of the programs, MDMX and CAX instructions were synthesized using annotations to instructions in the assembly files. The most time-consuming kernels were then identified by profiling, and the fragments of the baseline assembly language were manually replaced with ones containing MDMX and CAX

instructions. For a fair performance comparison of MDMX and CAX, additional instructions such as absolute differences accumulation and parallel conditional move were added to MDMX-type instructions, which are equivalent to the CAX instructions. Thus, MDMX (containing 30 instructions) and CAX (containing 34 instructions) have similar instructions, except for the permute instructions. Since the target platform is an embedded system, operating system interface code (e.g., file system access) was not included in this study. (Of course, the speedups of MDMX and CAX for complete programs may be less impressive than those for kernels due to Amdahl's Law [5].) The overhead of the color conversion was also excluded in the performance evaluation for all the versions. In other words, this study assumes that the baseline, MDMX, and CAX versions directly support YCbCr data in the same general data format (e.g., four-packed 8 bit |Unused|Cr|Cb|Y| for both baseline and MDMX and two-packed 6-5-5 bit |Cr|Cb|Y|Cr|Cb|Y| for CAX in a 32-bit register).

In addition, the Wattch-based simulator [2], an architectural-level power modeling, has been used to estimate energy consumption for each case. For the power estimates of the MDMX and CAX functional units (FUs), Verilog models of the baseline, MDMX, and CAX FUs were implemented and synthesized with the Synopsys design compiler (DC) using a 0.18-micron standard cell library. Power specifications of the MDMX and CAX FUs were then normalized to the baseline FU, and the normalized numbers were applied to the Wattch simulator for determining the dynamic power for the superscalar processor with MDMX or CAX.

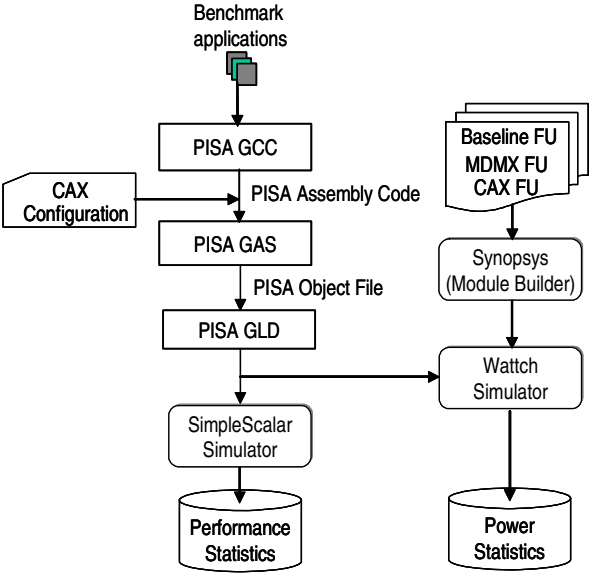


Fig. 4. A methodology framework for dynamically scheduled superscalar simulations

Table 2 summarizes the processor and memory configurations used in this study. A wide range of superscalar processors is simulated by varying the issue width from 1 to 16 instructions per cycle and the instruction window size (i.e., the number of entries

in the register update unit) from 16 to 256. When the issue width is doubled, the number of functional units, load/store queues, and main memory widths are scaled accordingly, in which the L1 cache (instruction and data) and the L2 cache are fixed at 16 KB and 256 KB, respectively. This study assumes that both MDMX and CAX use two logical accumulators, and all the implementations are simulated with a 180 nm process technology at 600 MHz and aggressive, non-ideal conditional clocking. (Power is scaled linearly with port or unit usage, and unused units are estimated to dissipate 10% of the maximum power.) With these processor configurations, we evaluate the impact of CAX on processing performance and energy consumption for the selected color imaging applications in the next.

Table 2. Processor and memory configurations

Parameter	1-way	2-way	4-way	8-way	16-way
Fetch/decode/issue/commit width	1	2	4	8	16
intALU/intMUL/fpALU/fpMUL/Mem	1/1/1/1/1	2/1/1/1/2	4/2/2/1/4	8/4/2/1/8	16/8/4/1/16
intALU/intMUL for MDMX and CAX	1/1	2/1	4/2	8/4	16/8
RUU (window) size	16	32	64	128	256
LSQ (Load Store Queue)	8	16	32	64	128
Main memory width	32 bits	64 bits	128 bits	256 bits	256 bits
Branch Predictor	Combined predictor (1 K entries) of bimodal predictor (4 K entries) table and 2-level predictor (2-bit counters and 10-bit global history)				
L1 D-cache	128-set, 4-way, 32-byte line, LRU, 1-cycle hit, total of 16 KB				
L1 I-cache	512-set, direct-mapped 32-byte line, LRU, 1-cycle hit, total of 16 KB				
L2 unified cache	1024-set, 4-way, 64-byte line, LRU, 6-cycle hit, total of 256 KB				
Main memory latency	50 cycles for first chunk, 2 thereafter				
Instruction TLB	16-way, 4096 byte page, 4-way, LRU, 30 cycle miss penalty				
Data TLB	32-way, 4096 byte page, 4-way, LRU, 30 cycle miss penalty				

4 Experimental Results

In the experiment, three different versions of the programs: (1) baseline ISA without subword parallelism, (2) baseline plus MDMX ISA, and (2) baseline plus CAX ISA are implemented and simulated using the Simplescalar-based simulator to profile the execution statistics of each case. The three different versions of each program have the same parameters, data sets, and calling sequences. In addition, energy consumption for each benchmark is evaluated using the Wattch-based power simulator. The dynamic instruction count, execution cycle count, and energy consumption form the basis of the comparative study.

4.1 Performance-Related Evaluation Results

This section presents the impact of CAX on execution performance for the selected benchmarks. The effect of loop unrolling for each program is also presented.

Overall Results. Figure 5 illustrates execution performance (speedup in executed cycles) for different wide superscalar processors with MDMX and CAX, normalized to the baseline performance without subword parallelism. The results indicate that CAX outperforms MDMX for all the programs in terms of speedup. For the 4-way issue machine, for example, CAX achieves a speedup ranging from 3× to 5.8× over the baseline performance, but MDMX achieves a speedup ranging from only 1.6× to 3.2× over the baseline. An interesting observation is that CAX exhibits higher relative performance for low-issue rates. For example, CAX achieves an average speedup of 4.7× over the baseline 1-way issue performance, but 3× over the baseline 16-way issue performance. This result demonstrates that CAX is an ideal candidate for multimedia embedded systems in which high issue rates and out-of-order execution are too expensive. Detailed performance benefits of CAX are presented next.

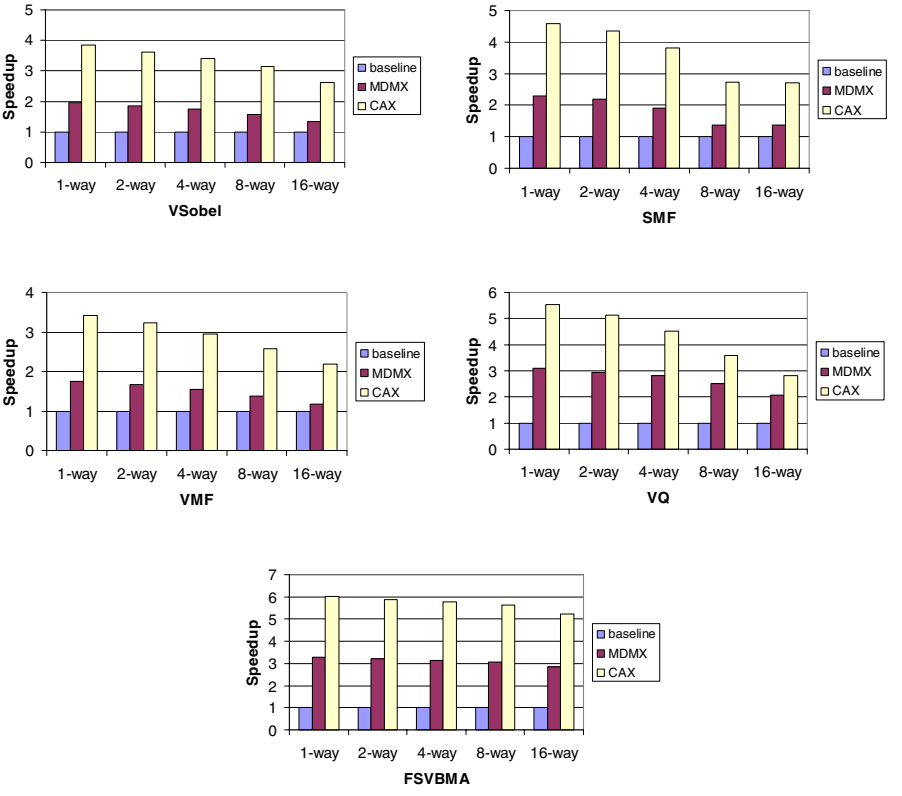


Fig. 5. Speedups for different issue-rate processors with MDMX and CAX, normalized to the baseline performance

Benefits from CAX. Figure 6 presents the distribution of the dynamic instructions for the 4-way out-of-order processor with MDMX and CAX, normalized to the baseline version. Each bar divides the instructions into the functional unit (FU, combines ALU and FPU), control, memory, MDMX, and CAX categories. The use of CAX instructions provides a significant reduction in the dynamic instruction count for all the programs.

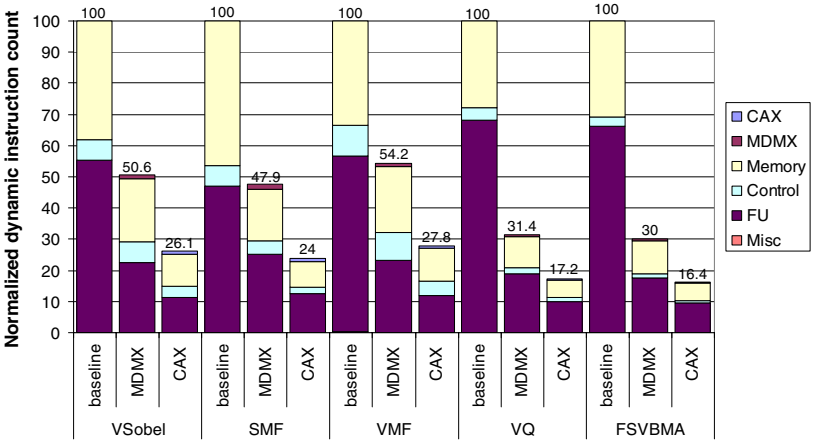


Fig. 6. Impact of CAX on the dynamic (retired) instruction count

Reduction in FU Instructions. The CAX arithmetic and logical instructions allow multiple arithmetic and logical instructions (typically three by processing three channels simultaneously) in addition to multiple iterations (typically two by processing two-packed YCbCr data) with one CAX instruction. Because of this property, all the programs using CAX reduce a significant number of the FU instructions and loop overhead, which increments or decrements index and address values. The reduction of the loop overhead further reduces the FU instruction count. Experimental results indicate that the FU instruction count decreases 73% to 86% (an average of 81%) with CAX, but only 47% to 73% (an average of 64%) with MDMX over the baseline version.

Reductions in Control Instructions. The CAX compare instructions allow multiple conditional (or branch) instructions with one equivalent CAX instruction, resulting in a large reduction in the control instruction count for all the programs. The control instruction count decreases 47% to 76% (an average of 60%) with CAX, but only 2% to 57% (an average of 26%) with MDMX over the baseline version.

Reductions in Memory Instructions. With CAX, multiple packed data are transported from/to memory rather than individual components. CAX accumulator instructions (e.g., `MACC_CRCBY` and `ADACC_CRCBY`) further eliminate memory operations since immediate results are stored in the accumulator rather than in memory. Experimental

results indicate that the memory instruction count decreases 68% to 83% (an average of 78%) with CAX, but only 37% to 66% (an average of 57%) with MDMX over the baseline version.

Overall, CAX clearly outperforms MDMX in consistently reducing the number of dynamic instructions required for each program. Performance improved by CAX is further enhanced through loop unrolling, which is presented next.

Benefits from Loop Unrolling. Loop unrolling (LU) is a well-known optimization technique that reorganizes and reschedules the loop body. Since loops typically contain the most critical code segments for color imaging applications, LU achieves a higher degree of performance by reducing loop overhead and exposing instruction-level parallelism (ILP) within the loops for machines with multiple functional units. Thus, the LU plus CAX technique may provide the much higher levels of parallelism and performance. Figures 7(a), (b), and (c) present an example of the inner loop of the BMA for vector quantization, the code after loop unrolling, and the loop from the perspective of CAX-level parallelism, respectively. The original loop is unrolled and reorganized through LU, shown in Figure 7(b). In the unrolled statement, multiple operands are then packed in each register with CAX, as shown in the dotted-line boxes in Figure 7(c). CAX then replaces the fragments of the assembly language for isomorphic statements grouped together in the dashed-line boxes with ones containing CAX instructions. Since operands are effectively pre-packed in memory, they do not need to be unpacked when processed in registers. In particular, the LU plus CAX technique provides the following benefits:

```
for (i=0; i<4; i++) {
   sum_y  += abs( IV_Y[i]  – CV_Y[i]);
   sum_Cb += abs( IV_Cb[i] – CV_Cb[i]);
   sum_Cr += abs( IV_Cr[i] – CV_Cr[i]);
}
```

(a)

```
sum_y  += abs( IV_Y[i+0]  – CV_Y[i+0]);
sum_Cb += abs( IV_Cb[i+0] – CV_Cb[i+0]);
sum_Cr += abs( IV_Cr[i+0] – CV_Cr[i+0]);
sum_Y  += abs( IV_Y[i+1]  – CV_Y[i+1]);
sum_Cb += abs( IV_Cb[i+1] – CV_Cb[i+1]);
sum_Cr += abs( IV_Cr[i+1] – CV_Cr[i+1]);
sum_y  += abs( IV_Y[i+2]  – CV_Y[i+2]);
sum_Cb += abs( IV_Cb[i+2] – CV_Cb[i+2]);
sum_Cr += abs( IV_Cr[i+2] – CV_Cr[i+2]);
sum_Y  += abs( IV_Y[i+3]  – CV_Y[i+3]);
sum_Cb += abs( IV_Cb[i+3] – CV_Cb[i+3]);
sum_Cr += abs( IV_Cr[i+3] – CV_Cr[i+3]);
```

(b)

```
sum_y  += abs( IV_Y[i+0]  – CV_Y[i+0]);
sum_Cb += abs( IV_Cb[i+0] – CV_Cb[i+0]);
sum_Cr += abs( IV_Cr[i+0] – CV_Cr[i+0]);
sum_Y  += abs( IV_Y[i+1]  – CV_Y[i+1]);
sum_Cb += abs( IV_Cb[i+1] – CV_Cb[i+1]);
sum_Cr += abs( IV_Cr[i+1] – CV_Cr[i+1]);
sum_Y  += abs( IV_Y[i+2]  – CV_Y[i+2]);
sum_Cb += abs( IV_Cb[i+2] – CV_Cb[i+2]);
sum_Cr += abs( IV_Cr[i+2] – CV_Cr[i+2]);
sum_Y  += abs( IV_Y[i+3]  – CV_Y[i+3]);
sum_Cb += abs( IV_Cb[i+3] – CV_Cb[i+3]);
sum_Cr += abs( IV_Cr[i+3] – CV_Cr[i+3]);
```

(c)

Fig. 7. (a) Original loop. (b) After loop unrolling. (c) CAX-level parallelism exposed after loop unrolling. IV and CV stand for the image and codebook vectors, respectively.

1. it reduces branch and address generation overhead,
2. it reduces register pressure and memory traffic by transporting multiple packed data from a register to memory and vice versa, and
3. it reduces a significant number of dynamic instruction counts.

Table 3 presents speedups for the baseline, MDMX, and CAX versions with LU, normalized to those without LU, in which the VSobel, SMF, and VMF programs were unrolled by a factor of 3; others were unrolled by a factor of 4. LU tends to be more effective for the CAX programs than the baseline and MDMX programs, indicating 21%, 4%, and 19% performance gains in the CAX, baseline, and MDMX versions, respectively. One of the major reasons is that LU reduces a similar number of loop overhead instructions for all three versions, but the total number of executed instructions for the CAX version is smaller than that for both baseline and MDMX versions. The next section presents energy-related performance since energy is as critical for embedded multimedia systems as performance.

Table 3. Speedups of the baseline, MDMX, and CAX versions with LU, normalized to those without LU

	VSobel	SMF	VMF	VQ	FSVBMA	Aver-age
Baseline plus LU	1.05	1.06	1.07	1.04	1.02	1.04
MDMX plus LU	1.24	1.23	1.28	1.14	1.09	1.19
CAX plus LU	1.27	1.24	1.29	1.16	1.10	1.21

4.2 Energy-Related Evaluation Results

Figure 8 presents the distribution of energy consumption for the 4-way out-of-order processor with MDMX and CAX, normalized to the baseline version. Each bar divides the energy consumption into the cache, ALU, clock, window, and others (combines branch prediction, rename, load-store queue, and result bus) categories. When execution platforms employ identical clock rates, implementation technologies, and processor parameters, a shorter execution time results in lower energy consumption [15]. Thus, CAX reduces a large amount of total energy consumption for all the programs due to a significant reduction in the executed cycle count. Experimental results indicate that CAX reduces energy consumption from 68% (VMF) to 83% (FSVBMA) over the baseline. This is in contrast to MDMX, which reduces energy consumption from only 39% (VMF) to 69% (FSVBMA) over the baseline. Since CAX reduces a large number of the ALUs, branches, and cache accesses, less energy is spent on the speculative execution and cache access units.

The energy consumption is further reduced with LU for all three versions of the programs, indicating an average energy reduction of 4.8%, 18.8%, and 19.2% for the baseline, MDMX, and CAX versions, respectively. In particular, LU reduces a large percentage of the power dissipation in the branch prediction hardware because it efficiently reduces branch overhead, indicating an energy reduction of 14.4%, 35.9%, and 36.3% in the branch prediction hardware for the baseline, MDMX, and CAX versions, respectively. Removing branches using LU also reduces the power dissipation in the fetch unit. The fetch unit fetches large basic blocks without being interrupted by

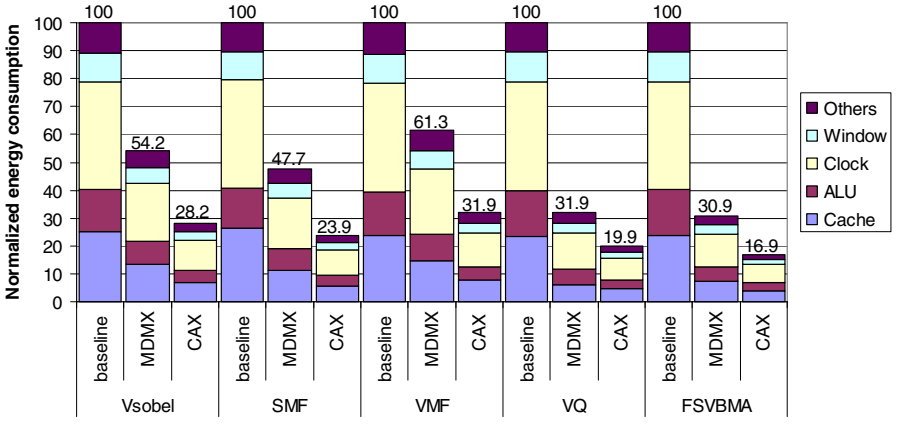


Fig. 8. Impact of CAX on energy consumption

taken branches, providing more work for the renaming unit and filling up the register update unit (RUU) faster. Thus, when the instruction queue and RUU are full, the fetch unit is stalled during the cycles. Because of this, the power dissipation of the fetch unit is reduced. Clearly, LU is effective at reducing additional energy consumption for image processing kernels where loop overhead is significant.

5 Conclusions

Color image and video processing has garnered considerable interest over the past few years since color features are valuable in sensing the environment, recognizing objects, and conveying crucial information. However, being a two-dimensional and three-channel signal, a color image requires increased computation and storage. To support these performance- and memory-hungry applications, a color-aware instruction set extension (CAX) has been presented that supports parallel operations on two-packed 16-bit YCbCr data in a 32-bit data processor, providing greater concurrency and efficiency for vector processing of color image sequences. Unlike typical multimedia extensions (e.g., MMX, VIS, and MDMX), CAX harnesses parallelism within the human perceptual YCbCr color space rather than depending solely on generic subword parallelism. In particular, the key findings are as follows:

- CAX achieves a speedup ranging from 3× to 5.8× (an average of 3.9×) over the baseline 4-way issue superscalar processor performance. This is in contrast to MDMX, which achieves a speedup ranging from only 1.6× to 3.2× (an average of 2.1×) over the same baseline 4-way issue superscalar processor.
- CAX also reduces energy consumption from 68% to 83%, while MDMX reduces energy consumption from only 39% to 69% over the baseline.
- Moreover, CAX exhibits higher relative performance for low-issue rates. For example, CAX achieves an average speedup of 4.7× over the baseline 1-way issue performance, but 3× over the baseline 16-way issue performance.

- Performance improved by CAX has been further enhanced by loop unrolling. LU provides an additional performance gain of 21%, 4%, and 19% for the CAX, baseline, and MDMX versions, respectively. This result suggests that the CAX plus LU technique has the potential to provide the much higher degrees of performance required by emerging color imaging applications.

In the future, we will perform an in-depth analysis of CAX with completed video processing applications (e.g., MPEG and H.26L). In addition, we will compare CAX with a wider ranger of the multimedia extensions, industrial as well as those proposed in academic research, while extending the datapath with 64 bits.

References

1. T. Austin and D. Burger: The SimpleScalar Tool Set. Version 2.0. TR-1342, Computer Sciences department, University of Wisconsin, Madison
2. D. Brooks, V. Tiwari, and M. Martonosi: Wattch: A framework for architectural-level power analysis and optimizations. Proc. of the IEEE Intl. Symp. on Computer Architecture (2000) 83-94
3. J. J. Dongarra and A. R. Hinds: Unrolling loops in Fortran. Software-Practice and Experience. vol. 9, no. 3 (1979) 219-226
4. R. C. Gonzalez and R. E. Woods: Digital Image Processing. 2nd Ed., Prentice Hall (2002)
5. J. L. Hennessy and D. A. Patterson: Computer Architecture: A Quantitative Approach. Morgan Kaufmann (2003)
6. J. Kim: Architectural enhancements for color image and video processing on embedded systems. PhD dissertation, Georgia Inst. of Technology (2005)
7. J. Kim and D. S. Wills: Evaluating a 16-bit YCbCr (6:5:5) color representation for low memory, embedded video processing. Proc. of the IEEE Intl. Conf. on Consumer Electronics (2005) 181-182
8. A. Koschan: A comparative study on color edge detection. Proc. of the 2nd Asian Conference on Computer Vision, vol. III (1995) 574-578
9. MIPS extension for digital media with 3D. Technical Report http://www.mips.com, MIPS technologies, Inc. (1997)
10. MMX^{TM} Technology Technical Overview http://www.x86.org/intel.doc/mmxmanuals.htm
11. H. Nguyen and L. John: Exploiting SIMD parallelism in DSP and multimedia algorithms using the AltiVec technology. Proc. Intl. Conf. on Supercomputer (1999) 11-20
12. A. Peleg and U. Weiser: MMX technology extension to the Intel architecture. IEEE Micro, vol.16, no. 4 (1996) 42-50
13. N. Slingerland and A. J. Smith: Measuring the performance of multimedia instruction sets. IEEE Trans. on Computers, vol. 51, no. 11 (2002) 1317-1332
14. J. Suh and V. K. Prasanna: An efficient algorithm for out-of-core matrix transposition. IEEE Trans. on Computers, vol. 51, no. 4 (2002) 420-438
15. V. Tiwari, S. Malik, and A. Wolfe: Compilation techniques for low energy: An overview. Proc. of the IEEE Intl. Symp. on Low Power Electron. (1994) 38-39
16. M. Tremblay, J. M. O'Connor, V. Narayanan, and L. He: VIS speeds new media processing. IEEE Micro, vol. 16, no. 4 (1996) 10-20

A Portable Doppler Device Based on a DSP with High-Performance Spectral Estimation and Output*

Yufeng Zhang[1,2], Yi Zhou[1], Jianhua Chen[1], Xinling Shi[1], and Zhenyu Guo[2]

[1] Department of Electronical Engineering, Yunnan University, Kunming, Yunnan Province, 650091, The P. R. China
yfengzhang@yahoo.com, zhouyikm@sina.com, {chenjh, shixl}@ynu.edu.cn
[2] Department of Electrical and Computer Engineering, The George Washington University, Washington, DC 20052, USA
zyguo@seas.gwu.edu

Abstract. A low-cost and high-performance portable Doppler blood flow analysis device, which is based on a digital signal processor (DSP) TMS320V549 (Texas Instruments), contains a $240 * 320$ LCD color graphic display module (Hantronix) and a thermal printer (Seiko Instruments), is developed in this study. The complex real-time autoregressive (AR) modeling is implemented in this device to estimate the time frequency representation of blood flow signals. Directional Doppler spectrograms are computed directly from the in-phase and quardrature components of the Doppler signal. Sampling frequency can vary among 5kHz, 10kHz, 15kHz, 20kHz and 25kHz to optimize the displaying dynamic range according to the blood velocity. In order to increase the display quality and provide more comprehensive information about the components of the blood flow profile, The Doppler spectrograms can be displayed in real-time on the LCD in 256 colors. They can also be printed out in 13 gray levels from the thermal printer for recording. The Doppler spectrograms computed by the AR modeling are compared with those by the STFT. The results show that this compact, economic, versatile bi-directional and battery- or line- operated Doppler device, which offers the high-performance spectral estimation and output, will be useful at different conditions, including bedside hospitals and clinical offices.

1 Introduction

Doppler ultrasound is widely used as a noninvasive method for the assessment of blood flow, both in the central and peripheral circulation. It may be used to estimate the blood flow, to image regions of the blood flow and to locate sites of arterial diseases as well as flow characteristics and resistance of internal carotid arteries [1-3]. Doppler devices work by detecting the change in frequency of a beam of ultrasound that is scattered from targets that are moving with respect to the ultrasound transducer. The Doppler shift frequency Δf_D is proportional to the speed of the moving targets:

* This work was supported by the Yunnan Science and Technology Council under the Grant 2002C002.

T. Srikanthan et al. (Eds.): ACSAC 2005, LNCS 3740, pp. 118 – 130, 2005.

$$\Delta f_D = \frac{2vf \cos\theta}{c} \quad (1)$$

where v is the magnitude of the velocity of the target, f is the frequency of the transmitted ultrasound, c is the magnitude of the velocity of the ultrasound in blood, and θ is the angle between the ultrasonic beam and the motion direction. Since the scatters within the ultrasound beam usually do not move at the same speed, a spectrum of Doppler frequencies is observed [1-3].

Diagnostic information is usually extracted from the Doppler spectrogram computed using the STFT in conventional Doppler devices. This is due to the computational efficiency and widespread availability of the STFT algorithm [1,2]. However, the STFT is not necessarily the best tool for analyzing Doppler blood flow signals [4]. It has a main shortcoming in the trade-off between time and frequency resolutions. To increase the frequency resolution, a longer time interval is required. Thus, the stationary assumption may not be valid. In addition, the spectral components occurring in a large interval will be smeared in the time domain, resulting in a decreased time resolution. To partly solve this problem, the AR modeling has been used as an alternative technique. Kitney and Giddens [5] stressed the best performance on spectral tracking and spectral resolution of autoregressive spectral estimation when short frames were used. Kaluzynski [6] reported the advantages of using the AR modeling for the analysis of pulsed Doppler signals, especially for short data lengths. Vaitkus et al. [7,8] addressed the good spectral matching ability of the AR modeling approach using a simulated stochastic stationary Doppler with a known theoretical spectrum as a reference to test spectral estimation techniques. Wang and Fish [9] used simulated Doppler signals to compare the performance of five signal analysis techniques, including the AR spectra estimation, and concluded that all the nonclassical methods had an improvement over the STFT for the bandwidth estimation.

The real-time sonogram outputs of the AR modeling and the STFT spectral analysis of data from 20MHz pulsed ultrasonic Doppler blood flow meter were presented by Güler et al [10]. Data obtained from coronary, renal, iliac, digital and mesenteric arteries were processed using AR- and FFT- based spectral analysis techniques and interpretable sonograms were constructed. In comparison with the FFT-based sonogram outputs, the AR-based sonogram outputs for 20 MHz Doppler data provide better results. Hence, the AR modeling was strongly recommended for small vessels with diameters between 1 and 2 mm. In another previous study, a system, implemented real-time AR modeling in a digital signal processor board for Doppler signal processing [11], was based on a digital signal processor and a microcomputer programmed to estimate the entropy autoregressive power spectrum of ultrasonic Doppler shift signals and display the results in the form of a sonogram in real-time on the computer screen. The results showed that the feasibility of on-line AR spectral estimation made this type of analysis an attractive alternative to the more conventional fast Fourier transform approach to the analysis of Doppler ultrasound signals. However the above two researches were conducted in the PC-based experiment system.

In present study, we design a small sized device based on a digital signal processor, a LCD graphic display module, and a thermal printer, make it suitable for flexible applications. The AR modeling algorithm in real-time is implemented for producing more comprehensive and accurate information about the components of the blood

flow profile. Doppler spectrograms are displayed in real-time on the LCD in 256 colors for increasing the display quality. They can also be printed out in 13 gray levels from the thermal printer for recording. This paper presents the development of the hardware, the implementation of the Doppler signal processing algorithms, the methods of displaying the Doppler spectrograms on the LCD and the printing out the Doppler spectrograms from the thermal printer, followed by results, discussions and conclusions.

2 Hardware

The hardware of the portable Doppler device consists of four basic parts: an analog board, a digital board, a thermal printer and its control circuit, and a $240 * 320$ LCD color graphic display module and its control circuit. Fig. 1 shows the block diagram of the analog board. The master oscillator operates at a constant frequency and drives the transmitting crystal of the probe via an amplifier. The returning ultra-

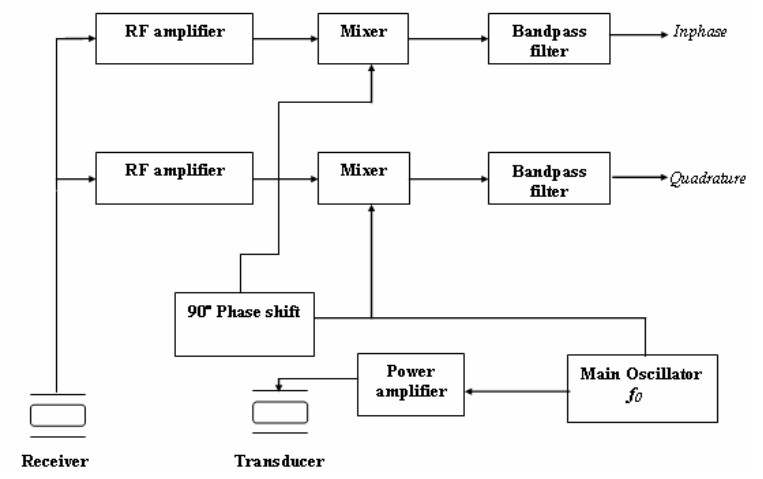


Fig. 1. The block diagram of the analog board of the portable Doppler device

sound signal, containing echoes from stationary and moving targets, is fed to the radio frequency (RF) amplifier from the receiving transducer. This amplified signal is then demodulated and filtered to produce audio frequency signals. The quadrature phase demodulation method, whereby the amplified signal is mixed with two quadrature reference signals (signals separated by a 90° phase shift) from the master oscillator, is used in this portable device [18]. This results in two audio signals called the in-phase and the quadrature Doppler signal respectively, both containing the Doppler information, but shifted by $\pm 90^{\circ}$ to each other, depending on whether flow is towards or away from the probe. The In-phase and the quadrature Doppler signals from this analog board are output to the digital board. Two kinds of ultrasonic frequency probes, 4 MHz and 8MHz, are equipped in this portable Doppler device.

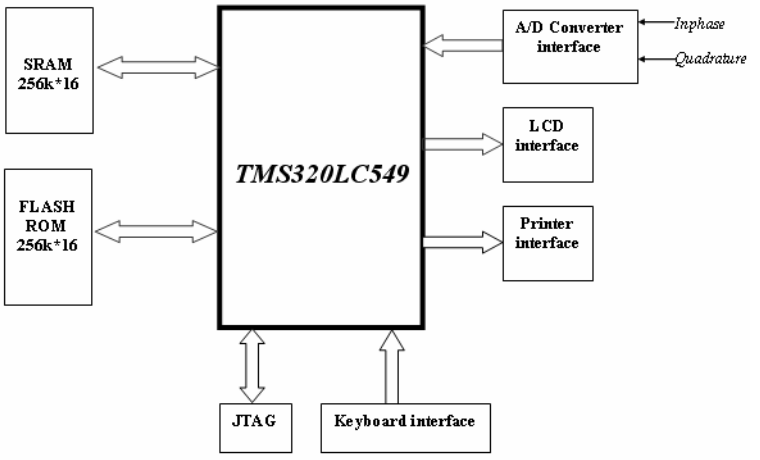


Fig. 2. The configuration of the digital board of the portable Doppler device

The digital board based on the digital signal processor TMS320V549 is capable of computing the complex STFT- based and the AR modeling- based spectra directly from In-phase and quadrature Doppler signals in real-time. The mean flow generated from spectra and displayed as separate forward and backward flow waveforms is smoothed for 15Hz equivalent bandwidth. Index values: resistance index (RI), pulsatility index (PI), maximum velocity (V_{max}), mean velocity (V_{mean}), which are calculated automatically from the average of the displayed waveforms in HOLD mode, are displayed and printed below the waveforms. The LCD display can scroll back 15 seconds of past waveform data stored in the 256k flash memory of the digital board. The block diagram of the digital board is shown in Fig. 2. The hardware of the digital board and algorithms are developed so that the signal can be received from the analogue-to-digital converter (ADC). The hardware accommodates the TMS320V549 DSP chip, address interface, SRAMs, FLASH ROMs, ADC circuit, real-time clock circuit, the LCD display interface circuit, keyboard interface circuit and the thermal printer interface circuit. This board includes 32k words of zero wait-state program RAM and 32k words of zero wait-state data RAM in DSP chip, 256k words of one wait-state data RAM, and a total of 256k words of flash ROM. The generic array devices, two chips of GAL20V8, select the RAM, FLASH ROM, or on board peripherals. In this configuration, the DSP can work at 80 MHz with two wait states when access the RAM.

The TMS320V549 synchronous serial port, whose interface consists of transmission and reception clocks, frame-sync and data lines, is used to access the onboard ADC TLV2548. This 12-bit CMOS analog-to-digital converter operates at 3.3v power supply, 2.8 us conversion time, and provides 8 analog inputs. It works in single-shot conversion mode. The auto-power-down mode is used for power-saving. A 4v built-in reference voltage is used for convenience. The converter uses the external 20 MHz serial input-output clock SCLK from the TMS320V549 as the source of the conversion clock to achieve high conversion speed. The sample-and-hold function is automatically started after the fourth SCLK edge. The sampling period is

programmed as short (12 SCLKs) to accommodate faster SCLK operation for high-performance signal processor TMS320V549. Sampling frequencies can vary among 5kHz, 10kHz, 15kHz, 20kHz and 25kHz to optimize the displaying dynamic range according to the blood flow velocity.

Implemented by using an EPSON SED1375 Embedded Memory Color LCD Controller, the Hantronix HDM3224C $240 * 320$ LCD color graphic display module with CCFL backlight is used in this portable device. The SED1375 is a color/monochrome LCD graphics controller with an embedded 80K Byte SRAM display buffer. The high integration of the SED1375 provides a low cost, low power, single chip solution to meet the requirements of the portable embedded device application. The SED1375 implements a 16-bit interface to the TMS320V549 digital processor, which may operate in Chip Select, plus individual Read Enable/Write Enable for each word mode. The SED1375 Look-Up Table (LUT) consists of 256 indexed red/green/blue entries. Each LUT entry consists of a red, green, and blue component. Each component consists of four bits, or sixteen intensity levels. Any LUT element can be selected from a palette of 4096 ($16 * 16 * 16$) colors. The SED1375 works at Eight Bit-Per-Pixel (256 Colors) mode. In this mode, one byte of display buffer represents one pixel on the display. When using a color panel, each byte of display memory acts as and index to one element of the LUT. The displayed color is arrived by taking the display memory value as an index into the LUT.

A Seiko LTP3445 Thermal Printer Mechanism with 112 mm paper width is used in the Doppler device. The printer drive electronics located on the digital board are based on the PT304P01/PT500GA1 Printer Chip Set. The printer control interface is equipped with a 8-bit parallel interface accessed by a 8-bit I/O port of the TMS320V549 digital processor. The print speed is 25 mm/sec using preprinted graph paper with 832 dots per line (8 dots/mm resolution). The printer operated at +7V (varies with battery) power supply uses the dynamic division to limit the total system peak current to approximate 3.4A while printing. The Doppler probe and other unused circuits are shut down while printing. The printer working at the LINE IMAGE printing mode prints out total 832 dots as a line on the graph paper from left to right corresponding to the 104 bytes data buffer from bit0 in the byte0 to bit7 in the byte103.

A 14 pin standard interface used by JTAG emulators to interface to TMS320V549 digital processor's On-Chip Scan-Based Emulation Logic, IEEE Std 1149.1† (JTAG) Boundary Scan Logic, simplifies code development and shortens debugging time. The chip DS1687, working at 3.3v power supply, embedded a 32.768 kHz crystal, and a lithium battery in a complete, self–contained timekeeping module, provides the real-time clock to the portable device.

3 Software Development

The software, which is programmed using TI C54x assembly language and TI C54x C language (Texas Instruments) [16] can be divided into two main parts: the data processing and the user interface. The main process is implemented as a main task loop. Interrupt-driven data acquisition is performed as a background task triggered by a sampling pulse generated by an embedded timer.

The in-phase and the quadrature Doppler signals are digitized by the 12-bit ADC with the sampling frequencies varied among 5 kHz, 10 kHz, 15 kHz, 20 kHz and 25 kHz according to the velocity measured. The digitized signals are then received by the TMS320V549 digital signal processor for processing to estimate the spectrum in real-time. The directional Doppler blood flow signal is a complex-valued signal expressed as

$$x(n) = x_i(n) + jx_q(n) \tag{2}$$

where $x_i(n)$ and $x_q(n)$ are the in-phase and the quadrature components respectively. In order to calculate the Doppler spectrogram in real-time, two data frame buffers are used to store the digitized Doppler audio signals. Each frame length is $2 * 256$ words. One buffer is used to store the current digitized signal frame acquired as the background task triggered by the sampling pulse generated by the embedded timer. As soon as this frame segment is full, a new spectral estimation for the data of this frame will begin. At the same time, the signal sampling is still continuing and the digitized signal will be stored in another frame buffer. For real-time execution, several assembly routines in the TI C54x DSPLIB (Texas Instruments) [17] are used to perform the signal processing. These routines include: CBREV --- complex bit reverse, CFFT --- forward complex FFT, and ACORR --- auto-correlation. All the computations are performed in the fast internal memory of the DSP chip. When a spectrogram estimation is finished, it will be displayed on the LCD in 256 colors in real-time. It can also be printed out in 13 gray levels from the thermal printer for recording when needed.

3.1 The STFT Algorithm

Consider a directional Doppler blood flow signal $x(t)$, and assume it is stationary when seen through a window $w(t)$ centered at time location τ, the Fourier transform of the windowed signal $x(t)w(t-\tau)$ is the short-time Fourier transform [13]

$$X_\tau(f) = \int x(t)w(t-\tau)e^{-j2\pi ft}dt \tag{3}$$

The STFT maps the signal from time domain into a joint time-frequency plane (τ, f). For implementation, the discrete version of (3) should be used as

$$DX_n(k) = \sum_{i=n-\frac{N}{2}}^{i=n+\frac{N}{2}} x(i)w(i-n)e^{-j\frac{2\pi k}{N}i} \tag{4}$$

where n and k are the discrete time and frequency, respectively, and N is the window length. The STFT-based time-frequency representation (energy distribution) of $x(i)$ is thus

$$SPEC(n,k) = \left|DX_n(k)\right|^2 \tag{5}$$

(4) and (5) are used to compute the Doppler time-frequency representation with a 10 ms Gaussian window ($\alpha = 3$) shifting every 10 ms. A 256-point FFT is computed for each windowed signal segment, resulting in a spectral frequency interval of $f_s/128$ Hz (f_s is the sampling frequency). Within one-second period, 100 spectra are computed.

3.2 The AR Modeling Algorithm

According to Guo et al. [12], it is possible to compute the forward and reserve blood flow components directly from the complex Doppler signal using complex AR modeling. The complex AR modeling is expressed as

$$x(n) = \sum_{m=1}^{p} a_{p,m}\, x(n-m) + e(n) \tag{6}$$

where *x(n)* is the complex Doppler signal, p is the order of the model, a_{pm} is the complex coefficients, and *e(n)* is the modeling error. The Yule-Walker Equations together with the Levinson-Durbin algorithm [11] are used to compute the complex AR coefficients a_{pm} and the modeling error variance σ_p. This algorithm proceeds recursively to compute the parameter sets $\{a_{1,1}, \sigma_1^2\}$, $\{a_{2,1}, a_{2,2}, \sigma_2^2\}$, ... , $\{a_{p,1}, a_{p,2}, \ldots, a_{p,p}, \sigma_p^2\}$, as follows:

$$a_{1,1} = -R_{xx}(1) / R_{xx}(0) \tag{7}$$

$$\sigma_1^2 = (1 - |a_{1,1}|^2) R_{xx}(0) \tag{8}$$

The recursion for *m*= 2,3, ... , *p* are given by

$$a_{m,m} = -\left[R_{xx}(m) + \sum_{k=1}^{m-1} a_{m-1,k} R_{xx}(m-k) \right] \Big/ \sigma_{m-1}^2 \tag{9}$$

$$a_{m,i} = a_{m-1,i} + a_{m,m} a^*_{m-1,m-i} \quad (i=1, \ldots, m\text{-}1) \tag{10}$$

$$\sigma_m^2 = (1 - |a_{m,m}|^2) \sigma_{m-1}^2 \tag{11}$$

where $R_{xx}(m)$ is the complex auto-correlation function of *x(n)* computed by

$$R_{xx}(m) = \frac{1}{N} \sum_{n=0}^{N-|m|-1} x(n+m) x^*(n) \tag{12}$$

$x^*(n)$ is the complex conjugate of *x(n)*. *N* is the number of data points. Once the AR coefficients are estimated, the power spectrum of the signal is computed using

$$S(k) = \frac{\sigma_p^2}{\left|1 + \sum_{m=1}^{p} a_{pm} \exp\left(-j \frac{2\pi k}{N} m\right)\right|^2} \tag{13}$$

The order of AR modeling is an important parameter to estimate the spectrum. In this study, fixed orders were used. According to Guo et al. [12], overestimating the order of the AR modeling introduces less error in the spectral estimation than underestimating it. However, too high a model order can introduce spurious peaks in the spectrogram. A model order between 5 and 16 should be used when a fixed order is selected to estimate each spectrum.

Equation (13) is used to estimate the AR-based spectrum with a frequency increment $\Delta f = f_s / 128$ Hz. Similar to the STFT, a time increment of 10 ms is used, and a total of 100 spectra are computed in one second.

3.3 The Methods of Displaying and Printing Out Spectrograms

As soon as a frame of the spectrogram computed by the DSP chip (using the STFT or the AR modeling algorithm) is completed, a subroutine service will be called immediately to display it on the LCD in 256 colors. In this subroutine, the DSP finds the square root of the power spectrum to reduce its dynamic range, and normalizes its value range from 0 to 255 corresponding to the LUT elements which consist of 256 colors. The power spectral density of the STFT and the AR modeling results are sequenced on the timeline to plot a three dimensional sonogram on the LCD. The color scales of sonograms represent the power levels corresponding to the frequencies at each point of time. As the power level is increased, the color tone of the sonogram goes into bright and as it diminishes, the color tone of the sonogram becomes dark. The display subroutine program is developed in assembly language because of the tight speed requirements to deal with the spectra in real-time.

The Doppler spectral computation and display subroutines are called once the frame data acquisition is completed in the main task loop. During the spectral computation and display period, the processor can still be interrupted by the programmable timer for the next frame data acquisition. Thus the time interval of the spectral computation and display performed by the DSP should be less than 10 ms, which is the window duration used to estimate the spectrum in this study because the signal is assumed to be stationary over this time segment.

In this application, the spectrograms can be printed out in 13 gray levels from the thermal printer for recording in HOLD mode to provide fast, complete and accurate documentation. This unique feature allows you to review up to 50 seconds of data and choose the ideal waveforms to document the patient's condition. Ordered dither halftoning technique is used to meet this demand. In this technique, a two-dimensional $6 * 2$ array is designed and the halftoning process is accomplished by a simple choosing the pixel corresponding to the gray scale level. This method is selected to process the spectrogram printing because it is straightforward and requires little computation. Depending on the progressive ordering of how halftone dots in a cell are turned on/off, ordered dither can be classified into clustered-dot and dispersed-dot [15]. In clustered-dot ordered dither, adjacent pixels are turned on/off as gray level

changes to form a cluster in the halftone cell. Clustered-dot dither is primarily used for printing devices that have difficulty when printing isolated single pixel. Obviously, this congregation of pixels will result in noticeable low-frequency structures in the output image. On the other hand, in dispersed-dot ordered dither, halftone dots in a cell turned on individually without grouping them into clusters. Therefore, sharp edges can be better rendered compared to clustered-dot dither. Considering the fact that the thermal printer used in present study is not sensitive to print isolated single pixel, the Clustered-dot dither mode is used to plot out the Doppler spectrogram. The 13 grey level halftone pixels with the clustered-dot ordered dither are shown in Fig. 3. When the printing subroutine is called, the DSP calculates the square root of the power spectrum estimated using the STFT or the AR modeling algorithm, and normalizes its value range from 0 to 12. The grey level of normalized spectrum represents the power level corresponding to frequency at each point of time. As the power level is increased, the gray level of the printed spectrogram goes into black and as it diminishes, the gray level of the printed spectrogram becomes light.

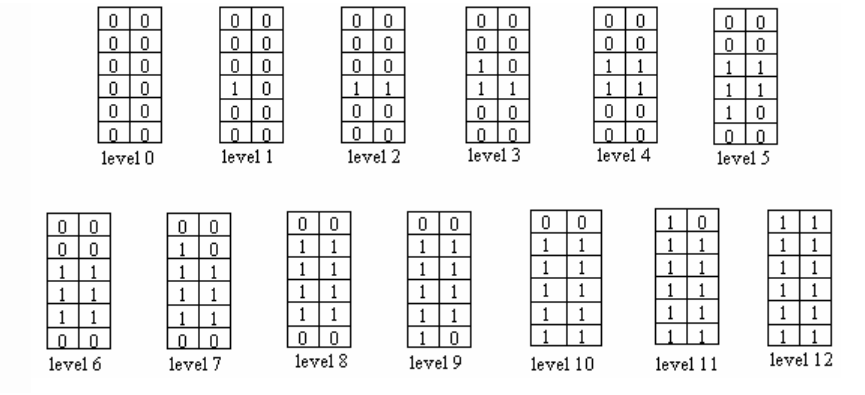


Fig. 3. The 13 grey level halftone pixels with the clustered-dot ordered dither

4 Results and Discussion

Fig. 4 shows the typical Doppler spectrograms of a normal femoral artery displayed on the LCD of the device in 256 gray levels based on the STFT and the AR modeling with the model order 15 respectively. Fig. 5 shows the typical Doppler spectrograms of a normal femoral artery printed from the thermal printer of the device based on the STFT and the AR modeling with different model orders p. There is a distinct qualitative improvement in the frequency spectra obtained using the AR algorithm with different model orders over the STFT. Especially, these figures show that the most important difference between using the STFT and the AR modeling algorithms is the Doppler speckles. The Doppler speckles based on the STFT are heavier than those based on the AR modeling because a pth-order AR modeling is constrained to have fewer than p spectral peaks. Comparing the AR- based spectrograms with different orders p=11, 15 and 19, the display quality has no significant difference.

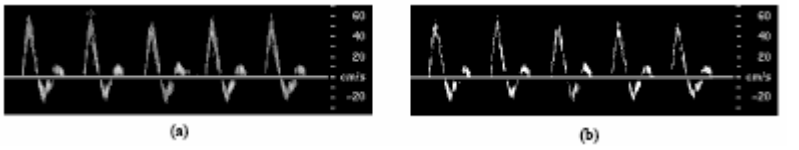


Fig. 4. The typical Doppler spectrograms of a normal femoral artery displayed on the LCD based on (a) The STFT at sampling frequency 25 kHz. (b) The AR modeling with p=19 at sampling frequency 25 kHz.

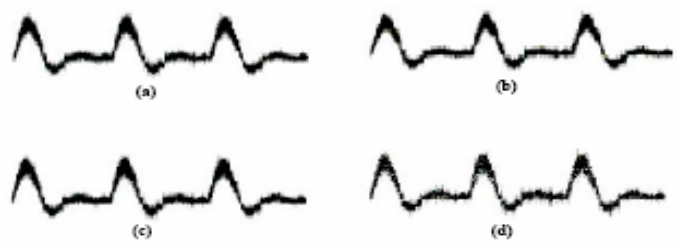


Fig. 5. The typical Doppler spectrograms of a normal femoral artery printed from the thermal printer. They are estimated from the signal at sampling frequency 25 kHz based on (a) The AR modeling with p=11. (b) The AR modeling with p=15. (c) The AR modeling with p=19. (d) The STFT.

The time intervals used to estimate the spectrum of individual frame based on the STFT and the AR modeling algorithms are given in Table1. In all cases, the time interval for each interrupt service routine T_i (used to process the data acquisition) should be multiplied by sample number N_s in a frame and added to the total processing time interval T_i when running in real-time. Thus

$$T_t = N_s * T_i + T_s + T_d \tag{14}$$

where T_s is the spectral estimation time interval, and T_d is the time interval used to display the spectrogram with 128 frequency components in a frame. In this application, T_d typically equals 2.1 ms. The spectral estimation time interval T_s is dependent on the spectral estimation algorithms. When using the STFT, T_s is 1.8 ms. However for the AR modeling algorithm, the spectral estimation time interval is dependent on the order of the AR model. For a model of order 5, the spectral estimation time interval T_s is 2.43 ms, for p=11 the time interval grows to 2.61 ms, and for p=19 to 3.11 ms. The time interval T_t is typically 3.71 us. For the STFT, when the maximum sampling frequency is 25 kHz, the samples in a frame are 250, the total time interval used by all interrupt routine services in a frame is 0.928 ms. However, in the case of AR modeling, the samples are 64 in a frame for all the sampling frequencies. The total time interval used by all interrupt routine services in a frame is 0.237 ms when using the AR modeling algorithm. The total processing time interval T_t of individual frame (including the time intervals due to the interrupt service routine and display the

spectrogram on the LCD) based on the AR modeling with different model orders and based on the STFT at different sampling frequencies are listed in Table 2 and Table 3 respectively. When using the STFT to estimate the spectrum, the total processing time interval T_t of individual frame is 4.828 ms at the maximum sampling frequency 25 kHz. While for the AR modeling with order p=19, the total processing time interval T_t of individual frame becomes 5.447 ms for all the sampling frequencies.

Table 1. The time interval T_s used to estimate the spectrum of individual frame based on the STFT and the AR modeling algorithms

Algorithm	STFT	AR modeling				
		p=5	p=9	p=11	p=15	p=19
T_s (ms)	1.8	2.43	2.54	2.61	2.81	3.11

Table 2. The total processing time interval T_t of individual frame based on the AR modeling algorithm with different model orders

Order p	5	9	11	15	19
T_t (ms)	4.767	4.877	4.947	5.147	5.447

Table 3. The total processing time interval T_t of individual framebased on the STFT algorithm at different sampling frequencies

Frequency (kHz)	5	10	15	20	25
T_t (ms)	4.086	4.271	4.457	4.642	4.828

Corresponding to the sampling frequency ranges of up to 25 kHz, this portable Doppler blood flow analysis device can process the Doppler shift frequency ranges of up to 12.5 kHz. With the TMS320V549 digital signal processor, it is possible to implement the Doppler spectrum computation and display it in 256 colors on the LCD module in real-time based on the STFT and the AR modeling of order up to 19 without losing data.

In this application, the TMS320V549 digital signal processor produces and displays the spectra in real-time at a rate of 100 frames per second. However this DSP is capable of running the present application at a rate of about 180 frames per second when using the AR modeling algorithm with the order p=19 at sampling frequency 25 kHz. If a better temporal resolution is desired, more frames should be calculated per second. In this case, another digital signal processor chip with higher processing speed should be considered.

5 Conclusion

The portable Doppler blood flow analysis device, which is based on a digital signal processor TMS320V549, contains a $240 * 320$ LCD color graphic display module and a thermal printer, is developed in present study. Directional Doppler spectrograms are computed directly from the in-phase and the quardrature components of the Doppler signal. The Doppler spectrograms estimated using the AR modeling can be displayed in 256 colors on the LCD module in real-time. They can also be printed out in 13 gray levels from the thermal printer for recording. The results based on the AR modeling are compared with those based on the STFT. This battery- or line- operated portable Doppler system, which produces crisp, clear Doppler sound complemented with a display and print-out to make documentation for reimbursement fast and convenient, is easy and simple to use for the evaluation of the peripheral vasculature. It has the advantage of being low-cost, high-performance, and will be useful at different conditions, including bed-side hospital and clinical office.

References

1. Cannon, S.R., Richards, K.L., Rollwitz, W.T.: Digital Fourier techniques in the diagnosis and quantification of aortic stenosis with pulse-Doppler echocardiography. J. Clin. Ultrasound 10 (1982) 101–107.
2. Kalman, P.G., Johnston, K.W., Zuech, P., Kassam, M., Poots, K.: In vitro comparison of alternative methods for quantifying the severity of Doppler spectral broadening for the diagnosis of carotid arterial occlusive disease. Ultrasound Med. Biol. 11 (1985) 435–440.
3. Young, J.B., Quinones, M.A., Waggoner, A.D., Miller, R.R.: Diagnosis and quantification of aortic stenosis with pulsed Doppler echocardiography. Amer. J. Cardiol. 45 (1980) 987–994.
4. Guo, Z., Durand, L.G., Lee, H.C.: Comparison of time-frequency distribution techniques for analysis of simulated Doppler ultrasound signals of the femoral artery. IEEE Trans. Biomed. Eng. 41 (1994) 332-342.
5. Kitney, R.I., Giddens, D.P.: Analysis of blood velocity waveforms by phase shift averaging and autoregressive spectral estimation, J. Biomech. Eng. 105 (1983) 398–401.
6. Kaluzynski, K.: Analysis of application possibilities of autoregressive modeling to Doppler blood flow signal spectral analysis, Med. Biol. Eng. Comput. 25 (1987) 373–376.
7. Vaitkus, P.J., Cobbold, R.S.C.: A comparative study and assessment of Doppler ultrasound spectral estimation techniques. Part I: Estimation methods, Ultrasound Med. Biol. 14 (1988) 661–672.
8. 8. Vaitkus, P.J., Cobbold, R.S.C.: A comparative study and assessment of Doppler ultrasound spectral estimation techniques. Part II: methods and results, Ultrasound Med. Biol. 14 (1988) 673–688.
9. Wang, Y., Fish, P.J.: Comparison of Doppler signal analysis techniques for velocity waveform, turbulence and vortex measurement: a simulation study. Ultrasound Med. Biol. 22 (1996) 635–649.
10. Güler, N.F., Kiymik, M.K., Güler, I.: Comparison of FFT- and AR- based sonogram outputs of 20 MHz pulsed Doppler data in real time. Comput. Biol. Med. 25 (1995) 383–391.
11. Schlindwein, F.S., Evans, D.H.: A real-time autoregressive spectrum analyzer for Doppler ultrasound signals. Ultrasound in Med. & Biol. 15 (1989) 263-272.

12. Guo, Z., Durand, L.G., Allard, L., Cloutier, G., Lee, H.C., Langlois, Y.E.: Cardiac Doppler blood-flow signal analysis: Part 2 Time frequency representation based on autoregressive modeling. Med. & Biol. Eng. & Comput. 31 (1993) 242-248.
13. Zhang, Y., Guo, Z., Wang, W., He, S., Lee, T., Loew, M. A.: comparison of the wavelet and short-time Fourier transforms for Doppler spectral analysis. Medical Engineering and Physics 25 (2003) 547-557.
14. Kay, S.M., Marple, S.L.: Spectrum analysis—a modern perspective, Proc. IEEE 69 (1981) 1380-1419.
15. Yu, Q., Parker, K.J.: Stochastic screen halftoning for electronic image devices. Journal of Visual Communication and Image Representation 8 (1997) 423-440.
16. TMS320C54x Assembly Language Tools User's Guide, Literature Number: SPRU102C 1998
17. The Staff of the Texas Instruments C5000 DSP Software Application Group, TMS320C54x DSPLIB User's Guide, 2001
18. Evans, D.H., McDicken, W.N.: Doppler Ultrasound: Physics, Instrumentation and Signal Processing. Second Edition, JOHN WILEY&SONS, LTD., 2000.

A Power-Efficient Processor Core for Reactive Embedded Applications

Lei Yang, Morteza Biglari-Abhari, and Zoran Salcic

Department of Electrical and Computer Engineering,
The University of Auckland, Private Bag 92019,
Auckland, New Zealand
{m.abhari, z.salcic}@auckland.ac.nz

Abstract. Reactive processors are a version of processors that provide architectural supports for the execution of reactive embedded applications. Even though much work has been done to improve the performance of reactive processors, the issue of optimizing power consumption has not been addressed. In this paper, we propose a new power-efficient processor core for reactive embedded applications. The new processor core (called ReMIC-PA) is implemented by adopting several power consumption optimizations to an existing reactive processor core (ReMIC). Initial benchmarking results show that ReMIC-PA achieves more than 20% power saving for data-dominated embedded applications and more than 50% power saving for control-dominated embedded applications when compared to ReMIC.

1 Introduction

Embedded applications can be classified into two categories: control-dominated and data-dominated applications [1]. In control-dominated applications, input events arrive asynchronously and the arrival time of these events is crucial. In data-dominated applications, on the other hand, input events arrive at regular intervals and the value of the events are more critical than the events' arrival time. Complex embedded applications always combine both categories.

Computer systems specified for handling reactive embedded applications are called reactive systems. They continuously react to input events fed by their environment through sending back output signals at a pace determined by the environment and they are purely input-driven. Reactive systems are usually implemented using conventional microprocessors or ASICs. However, conventional microprocessors are hardly matched to the performance requirements of most reactive applications, because their primary goal is to provide high data throughput instead of fast interaction. ASICs provide satisfactory performance, but at a price of high cost and little flexibility. To overcome limitations of the traditional approaches, reactive processors, which provide architectural supports for reactive embedded applications, have emerged in recent years [2, 3, 4]. They offer superior performance over conventional microprocessors and yet retain a high degree of flexibility, which is not achievable with ASICs.

T. Srikanthan et al. (Eds.): ACSAC 2005, LNCS 3740, pp. 131–142, 2005.

The idea behind reactive processors is inspired by Esterel [5], a synchronous reactive language specifically designed for reactive programs. Esterel provides a set of constructs for modeling, verification and synthesis of reactive systems. At present, most research activities related to reactive processors focus on improving the performance and less attention has been made in analyzing power consumption, which is obviously very important in many embedded applications. In this paper, our focus is on characterization and reduction of power consumption in reactive processors. We propose extensions to an existing reactive processor core [6] to make it more power efficient. The new processor core is implemented on an FPGA.

The main power consumption sources can be divided into two parts: static and dynamic power consumptions. Static power is dissipated when the logic-gate output is stable and it is dependent on the supply voltage and leakage current. Dynamic power is consumed during the switching activities of logic gates. Equation 1 indicates different factors involved in dynamic power dissipation.

$$P_{dynamic} = \frac{1}{2}.\beta.C.V_{DD}^2.F \tag{1}$$

where: β = Switching Activity per Node,
C = Switched Capacitance
F = Frequency (switching events per second)
VDD = Supply Voltage

FPGAs consume more static and dynamic power than ASICs and full custom ICs. For providing programmability, FPGAs contain more logic resources than custom integrated circuits for a certain amount of logic required by the design. Although the additional resources are not utilized in the design, they consume static power as long as the supply voltage is provided in current FPGA architectures. In FPGA devices, interconnects between logic blocks are programmable. These programmable interconnects have higher switched capacitance than the fixed interconnects so that they cause more dynamic power dissipation.

The main contributions of this paper are: (1) investigating the effects of an architectural support for reactive embedded applications from power dissipation point of view. For this purpose, an existent reactive processor core was examined. (2) employing power optimization techniques at the architecture level to improve the power dissipation of the reactive processor core based on the resources available in current FPGA technology.

The paper is organized as follows. In section 2, we outline the related works carried out in this research area. In section 3, we briefly introduce the initial base reactive processor architecture called ReMIC. The extensions made to ReMIC to make it power aware processor are discussed in section 4. This new processor is called ReMIC-PA. Experimental results, which demonstrate the features of the new processor in terms of power consumption, are presented in section 5. Conclusions are made in section 6.

2 Related Work

Power consumption of a system can be optimized at different levels of abstraction, ranging from the transistor level to the higher levels of abstractions including gate, register transfer, architecture and algorithm levels. This paper focuses on optimizations at the architectural level.

FPGAs have been used mostly as hardware accelerators in most applications for performance improvement with little attention to low-power design issues. This might be because of less demand for using FPGAs in power critical applications. Some research works on power optimization for FPGA architecture have been reported in [7, 8, 9]. In FPGAs, the efficiency of the design is also determined by the synthesis algorithms so that it is not easy to control the final synthesis result at the high level of abstraction.

Minimizing the switching activity of signals is the primary approach to reduce dynamic power dissipation in FPGAs at the architectural level. Reducing the switching activity of clock signals is an effective way to reduce overall switching activity of designs because clock signals switch continuously with higher capacitances. In [10], multiple clock signals are applied to the circuit so that the blocks on critical paths use higher speed clocks, whereas the blocks on non-critical or low-speed paths use lower speed clocks. In [11], a technique called clock gating is applied to the circuit to decrease the switching activity of clock signals. The concept behind clock gating is simple: there is no reason to clock the circuit if its output will not change state. Choosing the appropriate coding style can also reduce switching activity of signals. In [12], a technique called low-power encoding, which refers to choosing specific state encoding in finite state machines or value encoding in a counter to reduce switching activity of signals, is utilized to optimize dynamic power dissipation. Protecting circuits from glitches is another way to decrease the switching activity of signals. Glitches refer to momentary transitions that occur in combinational circuits due to delay imbalances in different gates. They can propagate to the next level of combinational logic as inputs and cause more transitions until the result is stabilized, or latches or registers are employed. There are two techniques to avoid glitches, retiming [13] and reordering [14]. The idea behind retiming is shuffling registers through the combinational block to reduce the depth of the combinational logic part of design. The reordering is through changing the order of input signals to reduce the glitches produced by output signals.

In our approach, we achieve power optimization in FPGAs by reducing the switching activity in the circuit.

3 ReMIC Architecture

ReMIC is a reactive processor core for handling reactive embedded applications. As illustrated in Fig. 1, it is developed based on a configurable processor core, called MiCORE [6] and extended with a functional unit to facilitate the execution of reactive embedded applications.

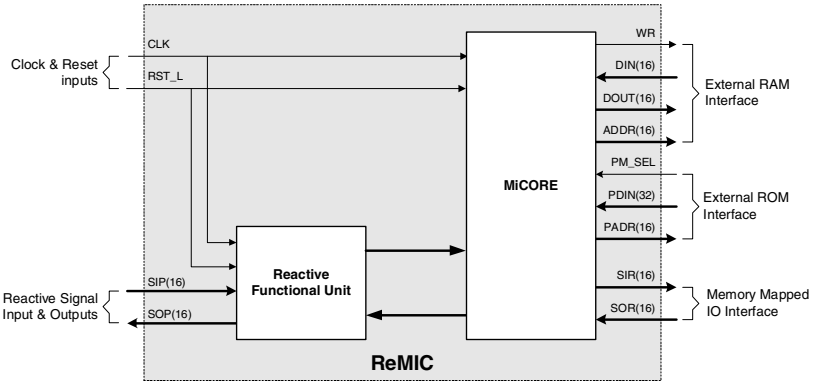


Fig. 1. ReMIC Block Diagram [6]

3.1 MiCORE

MiCORE is a 16-bit RISC-like pipelined configurable non-reactive processor core. It can be extended towards an application-specific processor by adding new instructions or by attaching external functional units. The basic features of MiCORE are as follows:

1. Two memory-mapped 16-bit I/O ports.
2. Five addressing modes: immediate, inherent, direct, register indirect and stack.
3. Three-stage pipeline architecture.

The MiCORE instructions have fixed and very simple format. All instructions are 32-bit long requiring one memory word. Each instruction is processed in three stages including instruction fetch stage (F), instruction decode stage (D) and instruction executing stage (E). Each stage takes one machine cycle. In stage F, a new instruction is fetched from the program memory location. In stage D, the effective data memory address is loaded into the address register according to the addressing mode. The specific operation is carried out in stage E. The detailed organization of the MiCORE data path and control unit can be found in [6]. In order to support configurability, the data path can be modified by adding or removing other processor resources, and the control unit is changed through providing proper sequence and type of micro-operations to support changes in the data path.

3.2 Reactive Extension

Polling and interrupt are two conventional ways provided by general-purpose processors to process control-dominated applications. The advantage of interrupt compared with polling is the speed of response to external events and therefore it avoids keeping the processor busy waiting for external events. However, interrupt

handling takes many processor cycles for context switching and execution of the interrupt handling routine. This becomes worse when priority and pre-emption are required.

ReMIC is implemented by extending MiCORE with a reactive functional unit (RFU) to assist execution of control-dominated applications. It follows the main ideas of Esterel, and adopts and supports Esterel model of reactivity at the instruction level. This architectural support results in a significant performance improvement compared to conventional processors which use interrupt mechanism [15].

ReMIC has a group of native instructions that support reactive processing. There are seven basic instructions in this group, which are presented in Table 1. All reactive instructions are 32-bit long and follow the standard MiCORE instruction format. EMIT and SUSTAIN are instructions to generate external signals which can last for one system clock cycle (EMIT) or continually (SUSTAIN). SAWAIT and TAWAIT present busy waiting mechanism on two types of events: external signal and time out (generated by internal timers). CAWAIT is a conditional polling mechanism to support branching after delay. ABORT is the preemption instruction and can handle priority.

Table 1. Native Reactive Instructions

Features	Instruction Syntax	Function
Signal Emission	EMIT *signal(s)*	Signal(s) is /are set high for one instruction cycle.
Signal Sustain	SUSTAIN *signal(s)*	Signal(s) is/are set high forever.
Signal Polling	SAWAIT *signal*	Wait until signal occurs in the environment.
Delay	TAWAIT *clock(s) prescaler*	Wait unit the number of timer clock cycles elapses. The basic timer clock is the system clock, but it can be adjusted using a prescaler.
Conditional Signal Polling	CAWAIT *signal1, signal2, address*	Wait until either signal1 or signal2 occurs. If signal1 occurs, then execute instruction from the next consecutive address, or else from the specified address.
Signal Presence	PRESENT *signal, address*	Instruction from the next consecutive address is executed if signal is present or else from the specified address.
Preemption	ABORT *signal, address*	If the signal occurs, the current instruction is completed and a jump to the specified address is made.

The RFU is implemented in a way to have very little dependency on MiCORE. The RFU can be easily removed from MiCORE or upgraded without modifying the MiCORE architecture. The detailed structure of the RFU can be found in [6].

The main responsibility of the RFU is handling ABORT instruction. ABORT instruction is introduced to perform preemption with priorities. In the current version, ABORT instruction can work with up to 16 different external input signals. Up to four levels of nesting of ABORTs is supported for handling priorities. ABORT instruction is executed through two stages:

1. ABORT Activation: When an ABORT instruction is fetched and decoded, it becomes active. The continuation address and the signal involved are stored in appropriate internal registers.
2. ABORT Termination: Once the designated signal is activated in the environment (provided it has a higher priority in nested ABORT blocks), ABORT is taken and an unconditional jump to the continuation address is executed (called pre-emptive ABORT termination), or if the continuation address is reached and the designated signal has not occurred in the environment, ABORT is automatically terminated (called non-pre-emptive ABORT termination).

4 Power-Efficient Implementation

ReMIC-PA is implemented through power-aware optimizations applied to ReMIC. The modifications target reducing dynamic power dissipation by minimizing switching activity of the design. The optimizations reported in this section can be classified into two parts targeting data-dominated and control-dominated applications respectively.

There are two optimization techniques which can be effective for the data-dominated parts of applications: (1) Precise Read Control (PRC) and (2) LiVeness Gating (LVG) [12]. The PRC is used to eliminate the register file reads based on instruction types. In the original ReMIC design, every instruction automatically reads two operands from register file in the decode stage no matter what instruction it is. This mechanism facilitates the proper pipeline operation at the cost of unnecessary reads. For example, instructions that operate on immediate values do not need to read the second operand from the register file. The LVG is responsible for elimination of the register file reads and pipeline registers updates when the pipeline is stalled or a taken branch instruction is executed. It is obvious that in the above two situations, both register file reads and pipeline registers update are worthless and should be eliminated. To support PRC and LVG, the extra circuitry used to control the register file access and pipeline register update is inserted into the original ReMIC control unit. The operation codes of instructions are also reordered according to the instruction types.

The optimizations for control-dominated applications are focused on minimizing the switching activity of the clock signal. ReMIC-PA provides a mechanism, which suspends the system clock when the core is idle and restores it when

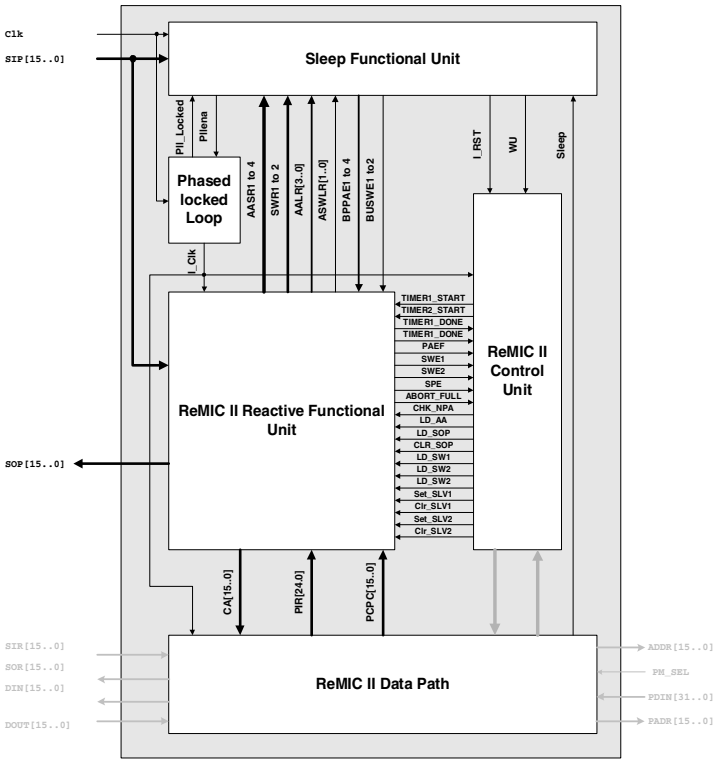


Fig. 2. ReMIC-PA Block Diagram

the designated input signals fed by the environment occur, to reduce power dissipated by the clock signal. To support this mechanism, a phased locked loop (PLL) and a functional unit called sleep functional unit (SFU) have been added to the original ReMIC. Fig. 2 illustrates the architecture of ReMIC-PA. The detailed description of ReMIC-PA can be found in [16].

4.1 Architectural Support for Power Optimizations in ReMIC-PA

ReMIC-PA provides three architectural supports for power optimizations:

1. Two executing modes, called the normal mode and the sleep mode respectively, which are implemented in ReMIC-PA. In the normal mode, the SFU enables the PLL to produce the system clock so that ReMIC-PA operates similar to ReMIC. In the sleep mode, on the other hand, the PLL is turned off by the SFU so that the system clock is gated and ReMIC-PA is suspended. The transition from the normal mode to the sleep mode is carried out by the execution of sleep mode related instructions. The restoration from the sleep mode to the normal mode is activated by the designated external input signals.

2. The SFU runs at a lower frequency than the processor core. As shown in Fig. 2. , the clock fed to the SFU is the same as the reference input clock of the PLL. Although the input clock frequency of the PLL can be an arbitrary value allowed by the FPGA device, in this case it is lower than the output clock frequency to reduce power dissipated by the SFU itself. For example, in Altera Cyclone devices [17], the minimum frequency of the reference input clock allowed for the PLL is around 16 MHz, and the maximum clock frequency of ReMIC-PA is around 50 MHz. Therefore, the SFU can achieve approximately three times less switching activity of the clock signal.
3. A set of power-efficient reactive instructions is provided by ReMIC-PA to support the optimizations.

4.2 Power-Efficient Instructions

ReMIC-PA has four power-efficient instructions that facilitate the mode transition. All instructions are presented in Table 2. They are 32-bit long and follow the standard ReMIC [15] instruction format.

Operations performed by the power-efficient instructions are almost the same as that of the corresponding reactive instructions, except that the reactive

Table 2. Power-effeicent Instructions

Features	Instruction Syntax	Corresponding Reactive Instruction	Function/Description
Power-efficient Signal Sustain	LSUSTAIN *signal(s)*	SUSTAIN	Bring the processor to the sleep mode and set signal(s) high forever.
Power-efficient Signal Polling	LSAWAIT *signal*	SAWAIT	Bring the processor to the sleep mode and wait unit the specified signal occurs in the environment.
Power-efficient Conditional Polling	LCAWAIT *signal1, signal2, address*	CAWAIT	Bring the processor to the sleep mode and wait until either signal1 or signal2 occurs. If signal1 occurs, the processor is restored to the normal mode and executes instruction from consecutive address; or else from the specified address.
Suspend	AWAIT	NONE	Bring the processor to the sleep mode.

instructions bring the processor in a wait state while the power-efficient instructions cause the processor to be suspended. These new instructions can be mixed with the other reactive instructions. When external input signals with shorter response deadlines are considered, the reactive instructions are used; otherwise, the power-efficient instructions are preferred.

5 Experimental Results

In order to compare improvements in power consumption, three processor cores are used for the experiments. These three processors represent a conventional processor core (IMIC), a processor core with instruction-level support for reactivity (ReMIC) and the optimized version of ReMIC for power consumption (ReMIC-PA). IMIC is based on MiCORE, which is extended to support interrupt mechanism. IMIC, ReMIC and ReMIC-PA designs are described in VHDL language at the RTL level and implemented in Altera Cyclone FPGAs.

The experimental results presented in this section are based on the set of reactive benchmark programs used in [2] in addition to four new programs for data-dominated applications. The data-dominated benchmark programs include a 16×16 unsigned multiplication, a fourth order FIR filter, an arithmetic averaging filter and an encoder from binary code to ASICII code. The control-dominated benchmark applications are initially written in Esterel and then manually mapped to IMIC, ReMIC and ReMIC-PA using their assembly languages. They include a car seat belt controller, a pump controller, a lift controller and a traffic light controller.

Table 3 presents the synthesis results using Altera Quartus 4.1. Obviously, the price paid for additional power efficient instructions and the SFU is larger number of logic elements (about 15% increase in using LEs in ReMIC-PA compared to ReMIC) and a little reduction of the maximum frequency (about 8% for ReMIC-PA compared to ReMIC).

First, we compare the power consumption for ReMIC and ReMIC-PA over the same data-dominated application programs. Table 4 indicates power consumption for each of these benchmarks when the system clock runs at different frequencies. Only dynamic power dissipated by internal logic elements is reported in the Table because the data-dominated optimizations presented in this paper

Table 3. Synthesis results for IMIC, ReMIC and ReMIC-PA

Altera Cyclon device - EP1C6Q240C6		IMIC	ReMIC	ReMIC-PA
Logic Elements	Number Used	1392	1472	1651
	Percentage Used	23%	24%	28%
I/O Pins	Number Used	150	164	164
	Percentage Used	81%	88%	88%
PLL Number	Number Used	0	0	1
	Percentage Used	0	0%	50%

Table 4. Comparison of power consumption of ReMIC and ReMIC-PA for data-dominated benchmark programs

Power Consumption (mw)		20 MHz	30 MHz	40 MHz
Multiplication	ReMIC	24.89	35.68	47.6
	ReMIC-PA	20.64	30.49	40.21
	Saving	19.6%	14.5%	15.5%
Averaging Filer	ReMIC	18.3	27.93	35.32
	ReMIC-PA	17.32	26.36	33.46
	Saving	5.4%	5.6%	5.3%
B2ASICII Encoder	ReMIC	27.37	43.46	53.2
	ReMIC-PA	14.55	21.84	29.03
	Saving	46.8%	49.7%	45.4%
FIR Filter	ReMIC	23.47	36.09	46.52
	ReMIC-PA	16.9	28.31	35.38
	Saving	27.9%	21.5%	23.9%
Average Saving		24.9%	22.8%	22.5%

Table 5. Comparison of Power Consumption of IMIC, ReMIC and ReMIC-PA for Control-dominated Benchmark Programs

Power Consumption (mw)		20 MHz	30 MHz	40 MHz
Car Seat Belt Controller	IMIC	10.99	16.03	21.03
	ReMIC	6.17	9.01	11.05
	ReMIC-PA	2.35	2.52	2.57
Elevator Controller	IMIC	15.84	23.58	32.07
	ReMIC	6.27	9.11	11.95
	ReMIC-PA	3.03	3.35	3.6
Pump Controller	IMIC	16.26	24.25	32.03
	ReMIC	6.27	9.21	12.05
	ReMIC-PA	4	4.61	5.07
Traffic Light	IMIC	20.18	30.06	40.09
	ReMIC	6.48	9.32	11.89
	ReMIC-PA	3.56	4.35	4.97
Average Saving (ReMIC-PA compared to ReMIC)		49%	59.6%	65.7%

are for minimizing the switching activity of the signals except the clock. Static power and dynamic power dissipated by the memory blocks and the global clock tree are excluded from this comparison as they were not the target for optimization. On average, ReMIC-PA achieves 23.4% power saving compared to ReMIC while executing these data-dominated benchmark programs. The average power saving of ReMIC-PA over the original ReMIC for data-dominated application benchmarks is almost similar for different system clock speeds.

In order to investigate the improvements in power consumption for reactive applications, the power consumption for IMIC, ReMIC and ReMIC-PA over

the same set of control-dominated applications are shown in Table 5. Since the control-dominated optimizations are focused on reducing the switching activity of the clock signal, dynamic power dissipated by internal logic elements and global clock tree is presented in the table. Static power and dynamic power dissipated by memory blocks are excluded. As shown in the Table, ReMIC-PA consumes much less power than ReMIC and IMIC for control-dominated benchmark programs. More power saving can be achieved as the clock speed increases. The shaded row (saving) in the Table indicates the saving in power consumption of ReMIC-PA compared to ReMIC.

6 Conclusions

In this paper, we propose a novel power-efficient reactive processor core, which is based on the ReMIC reactive processor architecture. Improvement in power consumption is achieved by adopting several optimizations to reduce power consumption for typical embedded applications. We synthesized and simulated the processor using Altera Quartus II 4.1 and implemented it on the Cyclone FPGAs. The power consumption of ReMIC and ReMIC-PA for a set of data-dominated applications; IMIC, ReMIC and ReMIC-PA for control-dominated applications are presented. For data-dominated applications, the ReMIC-PA achieved on average 23.4% power saving compared to ReMIC and the power saving only slightly depends on the system clock speed. For control-dominated applications, power saving achieved by ReMIC-PA is frequency dependent and varies between 49% (for 20MHz processor) to 65.7% (for 40MHz processor).

At present, ReMIC-PA can be mapped only to the Cyclone FPGAs since only the phased locked loops provided by Cyclone devices can be controlled by internal logic. In the other Altera FPGAs, the control signals of the phased locked loop have to be exported to external pins and therefore, board level support is required for the ReMIC-PA implementation.

The great improvement in power consumption of ReMIC-PA for control-dominated applications has a cost in resulting slower response time on an external event compared to ReMIC. The response time of the power-efficient instructions to a set of designated external events includes the restoration time of the PLL and the time spent on the handshake process between the SFU and the processor. In a Cylone FPGA device, the restoration time of the PLL is around 200 ns. If the SFU runs at 20 MHz and the clock frequency of the processor core is 40 MHz, the response time of the sleep related instruction is 200+50+25=275 ns. It is much longer than the response time of the reactive instruction, which is only one clock cycle or 25 ns in this case.

References

[1] Antoniotti, M., Ferrari, A., Lavagno, L., Sangiovanni-Vincentelli, A., Sentovich, E.: Embedded system design specification: merging reactive control and data computation. In: Proceedings of the 40th IEEE Conference on Decision and Control. Volume 4. (2001) 3302–3307

[2] Salcic, Z., Roop, P., Biglari-Abhari, M., Bigdeli, A.: Reflix: A processor core for reactive embedded applications. In Glesner, M., Zipf, P., Renovell, M., eds.: 12th International Conference on Field-Programmable Logic and Applications (FPL 2002). Volume 2448 of Lecture Notes in Computer Science. Springer-Verlag (2002) 945–954
[3] Roop, P.S., Salcic, Z., Biglari-Abhari, M., Bigdeli, A.: A new reactive processor with architectural support for control dominated embedded systems. In: proceedings of 16th International Conference on VLSI Design. (2003) 189 – 194
[4] Roop, P.S., Salcic, Z., Dayaratne, M.W.S.: Towards direct execution of esterel programs on reactive processors. In: proceedings of the 4th ACM International Conference on Embedded Software, Pisa, Italy (2004)
[5] Berry, G.: The Exterel v5 Language Primer. (2000)
[6] Hui, D.: A reactive micro-architecture with task level concurrency support. Master's thesis, Dept. of Electrical and Computer Engineering, University of Auckland (2004)
[7] George, V., Rabaey, J.M.: Low-energy FPGAs : architecture and design. Kluwer Academic publisher (2001)
[8] Poon, K.K.W., Yan, A., Wilton, S.J.E.: A flexible power model for fpgas. In Glesner, M., Zipf, P., Renovell, M., eds.: 12th International Conference on Field-Programmable Logic and Applications (FPL 2002). Volume 2448 of Lecture Notes in Computer Science. Springer-Verlag (2002) 312–321
[9] Cadenas, O., Megson, G.: A clocking technique with power savings in virtex-based pipelined designs. In Glesner, M., Zipf, P., Renovell, M., eds.: 12th International Conference on Field-Programmable Logic and Applications (FPL 2002). Volume 2448 of Lecture Notes in Computer Science., Springer-Verlag (2002) 322–331
[10] Papachristou, C.A., Nourani, M., Spining, M.: A multiple clocking scheme for low-power rtl design. IEEE Transactions on Very Large Scale Integration (VLSI) Systems **7** (1999) 266 – 276
[11] Xunwei, W., Pedram, M.: Low power sequential circuit design using priority encoding and clock gating. In: Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED '00). (2000) 143 – 148
[12] Petrov, P., Orailogulu, A.: Low-power instruction bus encoding for embedded processors. IEEE Transactions on Very Large Scale Integration (VLSI) Systems **12** (2004) 812 – 826
[13] Chabini, N., Wolf, W.: Reducing dynamic power consumption in synchronous sequential digital designs using retiming and supply voltage scaling. IEEE Transactions on Very Large Scale Integration (VLSI) Systems **12** (2004) 573 – 589
[14] Hashimoto, M., Onoedera, H., Tamaru, K.: Input reordering for power and delay optimization. In: Proceedings of Tenth Annual IEEE International ASIC Conference and Exhibit. (1997) 194 – 199
[15] Salcic, Z., Hui, D., Roop, P., Biglari-Abhari, M.: Remic - design of a reactive embedded microprocessor core. In: proceedings of ASP-DAC. (2005)
[16] Yang, L.: Low-power implementation of a processor core with architectural support for reactive embedded applications. Master's thesis, Dept. of Electrical and Computer Engineering, University of Auckland (2005)
[17] Altera: Cyclone Handbook, Chapter 6, Using PLL in Cyclone Device, *http* : *//www.altera.com/literature/hb/cyc/cyc*$_c$51006.*pdf*. (2004)

A Stream Architecture Supporting Multiple Stream Execution Models

Nan Wu, Mei Wen, Haiyan Li, Li Li, and Chunyuan Zhang

Computer School, National University of Defense Technology,
Chang Sha, Hu Nan, P. R. of China, 410073
meiwen@nudt.edu.cn

Abstract. Multimedia devices demands a platform integrated various functional modules and an increasing support of multiple standards. Stream architecture is able to solve the problem. However the applications suited for typical stream architecture are limited. This paper describes MASA (Multiple-morphs Adaptive Stream Architecture) prototype system which supports regular stream and irregular stream by different execution models according to applications' stream characteristics. The paper first discusses MASA architecture and stream model, and then explores the features and advantages of MASA through mapping a stream applications H.264 to hardware. The result is encouraging. At last, presents the design and implementation on FPGA of MASA's prototype.

1 Introduction

We are currently experiencing an explosive growth in development and deployment of multimedia devices that demands a platform integrated various functional modules and an increasing support of multiple standards. For example, a DVD platform consists of a logic DVD processor, an audio ADC, an audio DAC and a video CODEC. But just in video CODEC field, there are many popular standard formats such as WM V9, H.264, MPEG-2, MPEG-4, Real Video, DivX, H.263, H.263+, and VP6. Furthermore, these formats are developing rapidly, so it takes a risk to develop a commercial platform supporting only single standard format or application. This problem demands motivate the use of hybrid architecture which integrate a host processor and many special co-processor by inner buses such as sony's PSP Handheld Video Game System [1]. This solution takes advantage of mature technology to integrate system at lower business risk. However, hardwire special processors lack of flexibility and have high cost. Stream processor [2] fills the performance gap between special processor and general processor. It will become a better choice because that one chip can replace several special engines.

For media processing, the flexibility requirement points to the need of various communications, audio and video algorithms which differ in complexity. They have mostly a heterogeneous nature and comprise several sub-tasks with real-time performance requirement for data-parallel tasks [3][4]. A hardware which can cope with these demands needs different processing architectures: some are parallel, some are rather pipelined. In general, they need a combination. Moreover, various algorithms need different levels of control over the function units and different memory access.

T. Srikanthan et al. (Eds.): ACSAC 2005, LNCS 3740, pp. 143 – 156, 2005.

For instance, multimedia applications (as different video decompression schemes) may include a data-parallel task, irregular computations, high-precision word operations and a real-time component [5]. Conventional stream processor like Imagine [6] just supports SIMD execution model and regular stream access. It incurs that mapping some applications difficultly or completing inefficiently.

This paper presents MASA stream processor. It shares common characteristics of stream architecture and processes data stream. MASA provides three stream execution modes: single instruction multiple data (SIMD) model, multiple instruction multiple data (MIMD) model [7] and single kernel[1] multiple data (SKMD) model for the complex demands of media-processing applications. Of course, moderate hardware cost is inevitable. For instance, a *Reorder Cache* is used to reorganize stream on chip for irregular stream access. We choose h.264 encoder as an application and implement it on MASA simulator. The result is encouraging.

The remainder of this paper is organized as follows. Section 2 presents and describes MASA architecture. Section 3 discusses three stream executing models in MASA. Section 4 illustrates the computing process through mapping application onto MASA. Section 5 discusses the design and implementation on FPGA. The last section summarizes the conclusions drawn in this paper.

2 The MASA Architecture and Simulator

The prototype micro architecture of MASA is shown in Figure 1. MASA is a programmable stream processor, which works as a coprocessor. Scalar program is executed on the host processor. The MASA processor consists of 48 Arithmetic Pages (AP). Each AP in the array contains an ALU and a local register file (LRF) backed by banked stream register file (SRF). The processor follows stream execution model that one or more kernels are fetched and mapped onto the Arithmetic Page arrays.

2.1 Stream Memory System

Stream memory system transfers streams between software managed stream register file and off-chip SRAM or SDRAM. The memory system organizes data into stream. Several address generators and multiple data channels ensure the bandwidth between off-chip memory and stream processor.

2.2 Stream Controller

Scalar processor issues stream instructions to the stream controller. These instructions determine kernels' process. Stream controller dynamically schedules each stream instruction. According to the information from schedule unit, stream controller dispatches stream instructions to kernel controller.

The scoreboard consists of a pending instruction queue and the association logic to determine which instructions are for issue. The scoreboard will accept instructions from the host processor. It must first generate a resource mask which indicates which

[1] An application is decomposed to a series of computation kernels that deal with a great number of stream data in stream processing. A kernel is a small program.

hardware resources on the chip this instruction requires in order to execute. This mask is a collection of bits, each one representing a different hardware resource. Mask table is disparted into tow parts, one for APs and one for the other resources.

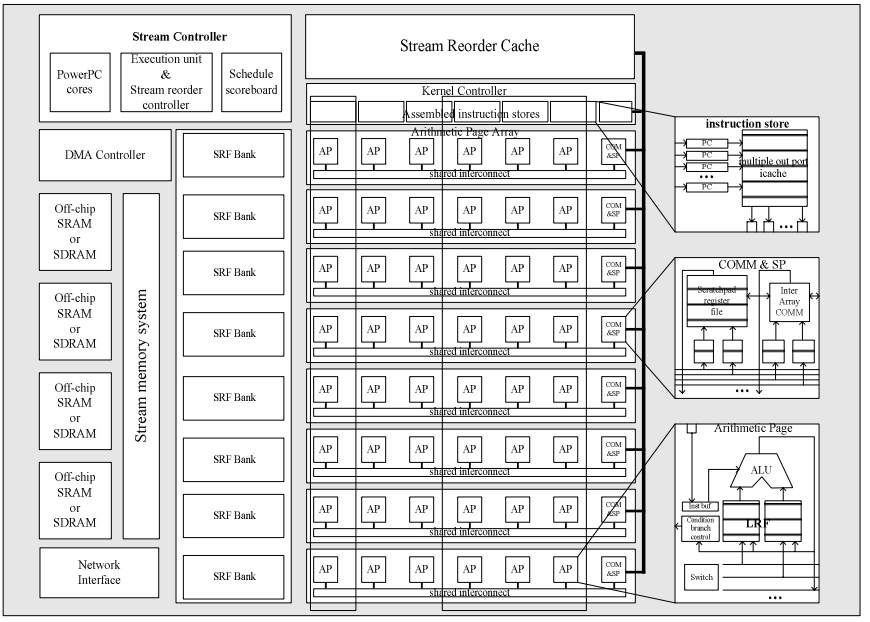


Fig. 1. Hypothetical, single-chip MASA system

Execution unit process the stream instruction and determines which Arithmetic Pages (AP) executes a certain kernel that means how to map the logic kernel to physical kernel execute units. It depends on the stream execution model that will be described in section 4. Each Arithmetic Page's status indicating busy or idle should be recorded. The information of this schedule is tightly related to the execution of two stream instructions that are load kernel and load data stream.

2.3 Stream Register File and Reorder Cache

Stream Register File (SRF) is responsible for the exchange of data between memory and kernel execution unit, and dataflow between kernels. It may work alone to keep regular stream access which is similar to the SRF used in Imagine, or work together with *Reorder Cache* to catch irregular stream access for the demand of various stream execution model.

There are stream buffers between memory at this level and local register file or Reorder Cache for high throughput of stream channel. Although the SRF is single-ported, applications require access to multiple streams simultaneously. Therefore, the SRF port is time-multiplexed over several streams, and stream buffers are used to match the access characteristics of the SRF to that of the computational kernels as shown in Figure 2(a). On each SRF access, N x m words are transferred

to/from the SRF from/to a single stream buffer. The compute clusters, on the other hand, access the stream buffers N words at a time (one word per cluster), but may access multiple stream buffers at once. A separate set of stream buffers are used to mediate transfers between the SRF and the memory system. Address generation for the SRF is performed by counters that keep track of the next block to be accessed for each stream. Arbitration among streams for access to the SRF port is dynamic and decoupled from kernel execution.

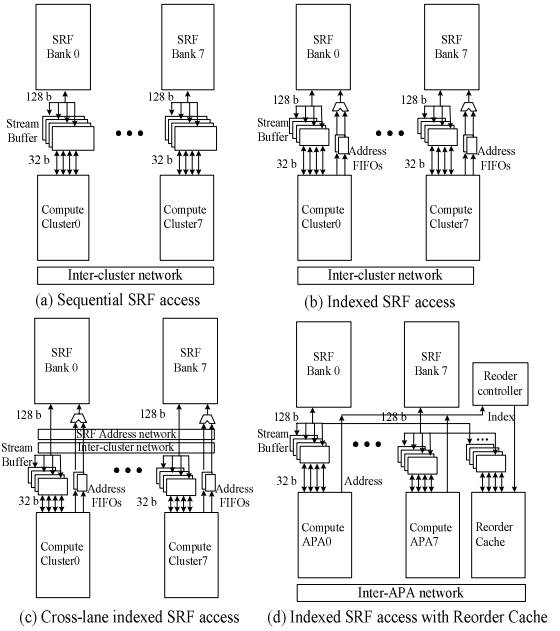


Fig. 2. SRF access mechanisms

Several sub-classes of data-parallel application domains such as 2D and 3D signal processing, cryptography, and scientific computing exhibit data reuse patterns that are not amenable to repeated accesses to long vectors or streams in a single order. The sequential access restriction of Stream register file presents an artificial impediment to extending vector or stream programming models to these applications that are otherwise promising candidates. So that the irregular stream access is required. Figure 2(b) (c) shows tow approaches of indexed SRF access mechanisms in Imagine [8]. In our opinion, these approaches do not emphasize irregular stream access but regular stream access. The mechanism described in Figure 2(b) limits indexed access from a cluster to the SRF bank in its lane. This level of indexing, coupled with statically scheduled inter-cluster transfers, is sufficient to support data reordering patterns that can be analyzed at compile-time. However, it is inefficient for data-dependant access patterns since any static schedule must allocate communication resources assuming worst-case conflicts and the data access crossing lane needs huge quantities of communication [8]. Cross-lane indexed access shown in Figure 2(c) allows any cluster to access any SRF bank with dynamic conflict resolution. However, this makes SRF

nearly the same as a multi-port cache, which weakens the advantage of stream memory such as scalability and low latency. And another drawback is that both these mechanism can not support SKMD and MIMD described in section 4, since it is lack of a decoupled controller to provide multiple kernels with respective streams simultaneously. MASA introduces a Reorder Cache (shown in Figure 2(d)) to solve the irregular stream and indexed scalar constants (such as 2D or 3D array) access. Reorder Cache simply works as a stream client of SRF as well as clusters. It loads streams which need reorder from SRF, and then store the stream that reordered by index back to SRF. This procedure is quit transparent to clusters that they load stream elements form SRF as usual. Clusters just generate stream access addresses to the reorder controller, reorder controller combine addresses to an index and sent it to the Reorder Cache. Also Reorder Cache directly connected to clusters by inter-cluster network to be more flexible.

2.4 Kernel Execution Mechanisms [7]

Kernel execution mechanism which contains kernel controller and 8 (may scale to more than 8) Arithmetic Page Arrays (APA) are basic units for executing kernel's microcode program. A single kernel controller controls multiple Arithmetic Page Arrays. It contains multiple assemble instruction stores, which respectively belongs to one column of function unit. Each instruction store consists of several program counters (PCs) and a multiple-issue ports instruction cache whose size is enough to store microcode of one kernel at least. Each PC corresponds to one Arithmetic Page in this column. Different instruction which PC points at can be issued to every Arithmetic Page in this column from multiple-issue ports. Multiple instruction stores can be aggregated together to issue one VLIW instruction. Each Arithmetic Page Array includes multiple Arithmetic Pages, one scratchpad register file and one inter-array communication unit in one row. Each Arithmetic Page consists of instruction buffer, decoder, condition branch control and one ALU. The condition branch control is used to control the numbers of loop iterations.

3 Stream Execution Model

MASA supports three modes of execution. Each of models is well suited for a different type of parallelism.

3.1 SIMD (Single Instruction Multiple Data) Model

Kernels are executed time-multiplexed in MASA. All instruction stores would be aggregated together to issue one VLIW instruction. During kernel execution, the same VLIW instruction is broadcast to all eight APAs in SIMD fashion. Stream operations are performed by passing streams from and back to the SRF.

MASA in this model is similar to Imagine. So all applications suited for Imagine are also suited for MASA. Time-multiplexing kernels with regular stream, especially load imbalance, are well-suited for the SIMD model.

3.2 MIMD (Multiple Instructions Multiple Data) Model

Kernels are executed space-multiplexed in MASA. Multiple kernels share the whole computing recourse at the same time. One or more columns of APs perform one kernel. APs in one row operate on one record of a stream so that one kernel is executed in SIMD fashion. While multiple columns of APs execute a kernel, instruction stores of these columns are aggregated to issue VLIW instruction. The output record produced by one kernel will be sent to the kernel consumed it directly. In other words, the intermediate stream is directly transferred between APs through high speed programmable switch in APA. Multiple columns of APs are allocated to the kernel in which the amount of computation is larger, if the speed of producing records and that of consuming records are not matched. The scratchpad register file is used to store LRF spilling. The SRF or reorder cache is required when the scratchpad register file is full.

In some "tile-based" stream architectures [9] such as RAW [10] and TRIPS [11], fine-grain MIMD is supported by allowing data transferred horizontally and vertically in ALU array. Therefore data transfer from one corner to another corner is inevitable. In Imagine, the intermediate stream between kernels has to be flow through SRF. However in MASA the interconnection of intra-APA provides high bandwidth of horizontal data transfer to ensure the inter-communication of kernels. In addition, it can reduce the bandwidth requirement for the SRF and take full use of the producer-consumer locality.

Load balanced kernels with regular stream or irregular stream, especially real-time demand are well-suited for the MIMD model. Of course, MASA limits vertical data transfer that can be done through inter-APA communication and SRF.

3.3 SKMD (Single Kernel Multiple Data) Model

All APs execute the same kernel where the instructions in the different processing elements are not synchronized at the instruction level. Each instruction store keeps the same kernel microcode. Each APA operates on one record of a stream. The amount of computation could be different from one record to another record such as vertex skinning [12, 14]. Each APA completes the loop of one record. Then reads the next record and executes the same kernel. Though every AP execute the same loop, the instructions in different APA could be are not synchronized at the iteration level and the instruction level. The same kernel is stored in different instruction stores, but different instructions can be issued to different APAs. The instruction issued to APA is determined by program counter respectively. Condition branch control in APA store initially stores the number of iterations of kernel executed in one record or loop end condition. When the iteration completes, the value is decremented. If the counter reaches zero or the end condition is satisfied (the condition can be dynamically determined by computed result), the processing of one record is completed. Then the next record of input stream is read and processed. Compare to SIMD model, it will not result in APA's idle and computation resource's waste in SKMD model.

For SIMD, all APAs consume stream elements at the same rate. So a stream buffer between the SRF and LRF which stores a stream temporarily can be read or writen by

all corresponding APAs and stream element can be prefetched. It is the nature of regular stream access. However, while different APAs consume stream elements at the different rate, prefetch is impossible. So stream memory can not work in old fashion. It maybe organized as FIFO queue or stack. APAs may consume stream records in the same SRF bank that a random-access cache is needed to reorder stream. In this case, the stream is loaded in the Reorder Cache first. Then each APA reads the next record from the cache instead of the SRF through inter-APAs network.

Both instructions and data are transferred through the same stream channel, but in different time, so the execution of an application is divided into two parts: configuration time and execution time. Mapping kernels into kernel execution mechanism according to *model* and loading kernels are completed in configuration time. After that, stream controller allows for transferring data and starting kernels. During this period, kernel is kept unchangeable in the assemble instruction store and data streams are transformed from SRF to APAs in sequence.

4 Application Study

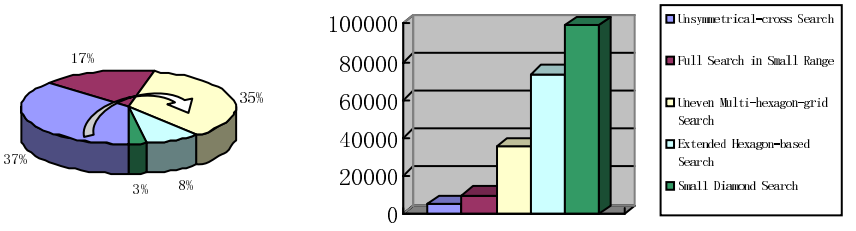
The stream programming model in MASA is similar to that in Imagine [13]. An application is decomposed into a series of computation kernels that operate on data streams. A kernel is a small program executed in Arithmetic Page Arrays that is repeated for each successive element of its input streams to produce output stream for the next kernel in the application. MASA can be programmed at two levels: stream-level (using StreamC) and kernel-level (using KernelC). Program at stream level controls the whole flow of a stream application's execution. The flow route of stream among kernels is similar to dataflow graph. At the kernel level, programmer is absorbed in the functional implementation of each kernel. More details about the programming model refer to [14].

Video compression is an important application domain in media processing. There are many popular formats. H.264 is a mainstream standard. Comparing to MPEG, data compression ratio using h.264 standard is higher at the cost of computation complexity. We implemented the key parts of h.264 encoder on the streaming processor with 8 Arithmetic Page Arrays to investigate their performance and functional unit utilization, including motion estimation, transform & quantization, entropy coding. According program characteristics, we map applications on different execution models and evaluate their performance. The window's size is 24x24 and the macro block's size is 8x8 in the whole application of this paper.

4.1 Motion Estimation

Motion Estimation is processed between current macroblock and reference macroblock in order to find the best match which has minimum SAD. Then the optimum motion vector is produced as the output. Motion estimation is the most time consuming part in the H.264 coding framework, nearly 60-80% of the total encoding time of the H.264 codec. UMHexagonS algorithm [15], adopted by H.264 formally, can decrease 90% of computation compared with full search algorithm. It uses a hybrid and hierarchical

motion search strategies, including five steps: 1) Unsymmetrical-cross search; 2) Full search in small range; 3) Uneven multi-hexagon-grid search; 4) Extended hexagon-based search; 5) Small diamond search. Considering early termination strategy, if search processing is able to meet the need of video definition that means computed SAD could satisfy preset-threshold, the current search step is enough. Otherwise, it is necessary to go on the following search steps tailed to higher definition. Figure 3(b) illustrates that there is great gap among the amount of computation till different steps. According to our statistical results of a given picture, we get the corresponding proportion for macro block when the search completes, shown in Figure 3(a). It is seen that most macroblock can achieve satisfying matching result after the foregoing three steps.



(a) Percentage for various search steps (b) #instructions for various search steps

Fig. 3. Characteristics of various search steps

We map UMHexagonS algorithm on to our MASA simulator by four different modes shown in Figure 4 and compare their implementing efficiency.

1) SIMD + macroblock stream + single kernel

This search kernel has two input streams-current macroblock (MB) stream and search window stream(SW), and one motion vector stream as its output stream, shown in Figure5(a). Each APA processes one macroblock in SIMD manner. All the operations in each APA are the same, so the five search steps of UMHexagonS algorithm must be implemented for one macroblock. As a result, each APA would execute all 99200 instructions. It brings waste to hardware resources up to 75.27%.

2) SIMD + macroblock stream + multiple kernels

In order to avoid the waste of computational ability in the first implementing mode, we organize different kernel for every search step. And stream-level program decides whether to execute the next kernel. The data diagram is shown in Figure 4(b). Stream organization is the same to that of the first mode, so each APA also processes one macroblock. The problems of this mode are: the disorder output of motion vector; short stream of filtered macroblock, even the waste of computing power if there is less than eight macroblocks to process till last step; increasing bandwidth requirement and switch cost among multiple kernels.

3) SIMD + pixel stream + multiple kernels

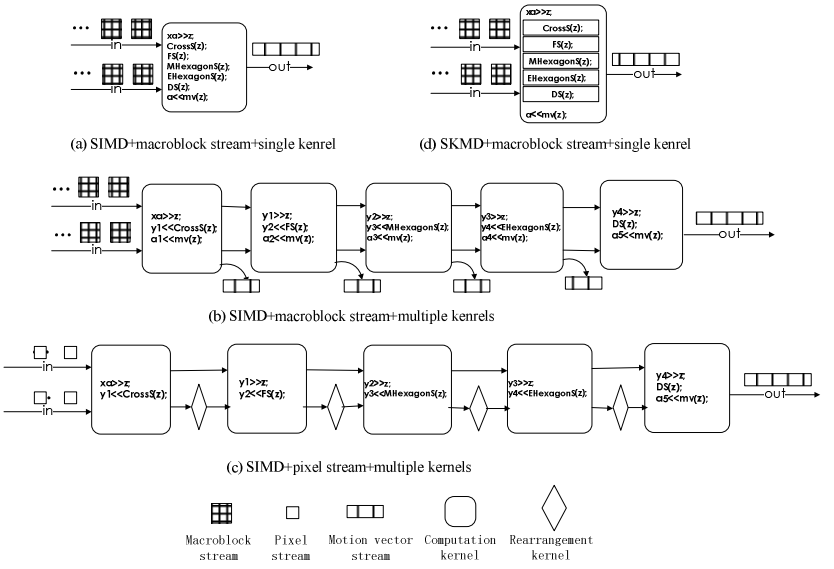


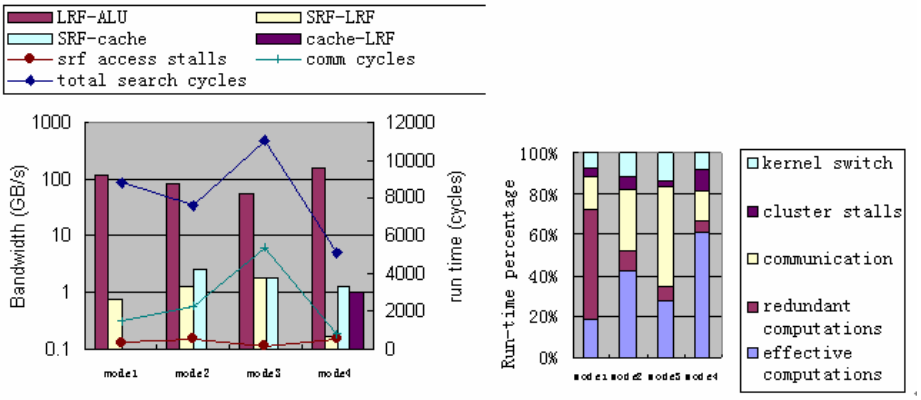
Fig. 4. Data stream graph

To solve these problems in the second mode, we organize input/output stream in the unit of pixel, and luma of 4 pixels as one 32-bit stream record. In this condition, search window may be partitioned into three rows, and each is organized by macroblock form left to right [18].

Figure5(c) illustrates its data diagram. Similar to the second mode, each kernel consist of different search step and stream-level program controls the implementation of next kernel. The difference is that all APAs process one macroblock together while each APA computes one record. But the search range of the latter kernel is determined by the result of its former. Large communication is the shortcoming of this mode because of the fine-grain stream record, including communication kernel cost and computing communication. SIMD may solve this problem by copy the records, such as coping the same window in scratchpad of each APA. However, it brings large redundancy for the bandwidth of SRF and LRF. As a result, the whole efficiency will come down.

4) SKMD + macroblock stream + single kernel

Combined with our proposed SKMD mode, we map the application in way of Figure 4(d). Stream record is a macroblock and each APA processes one macroblock. APAs support the execution of different macroblocks in different steps. When APA finishes one macroblock, it can load the next macroblock from reorder cache. Search window of each macroblock needs to be stored in scratchpad of its corresponding APA. And it can be accessed by index without redundancy. This mode can increase the ratio of effective computation. There is no communication cost because of inter-macroblock independence. And a polymeric single kernel avoids the switch cost of kernel. Thus, LRF bandwidth can be utilized fully and SRF requirement will be lower. As a result, we achieve higher efficiency. It can be seen that this mode has some characteristics of TLP. However, the processing time of each macroblock differs, so it brings the disorder of motion vector stream. it is required to rearrange records by extra operations.



(a) Bandwidth hierarchy and search time (b) Execution time breakdown

Fig. 5. mode 1: SIMD + macroblock stream + single kernel
mode 2: SIMD + macroblock stream + multiple kernels
mode 3: SIMD + pixel stream + multiple kernels
mode 4: SKMD + macroblock stream + single kernel

We experiment above all modes on MASA simulator. Figure 5 shows our experience data. In Figure 5(a), the left y-axis is bandwidth hierarchy and the right y-axis is run time of 8 macroblocks. We can easy find that mode4 takes full use of LRF bandwidth and achieves 2.3X speedup to mode1. The performance's difference of four modes is due to several factors shown in Figure 5(b). Besides of kernel switch, rest categories account for kernel run-time in the APAs. Kernel switch includes loading microcode and preparing streams etc. Obviously, redundant computations of mode1 and communications of mode3 cause the performance degradation.

4.2 Transform and Quantization

Butterfly calculation is a fast algorithm for the integer transform [16]. It can use less addition and shift to obtain the result of matrix multiplication. Integer transform $Y=C_fXC_f^T$ includes two matrix multiplications. Here, assume that $B=C_fX$, then $Y=C_fXC_f^T=BC_f^T=(C_fB^T)^T$. Based on transpose we keep the block that is to be transformed as right matrix while the left matrix is C_f. The matrix X can be divided into four vectors by column. Inter-column independence makes the butterfly algorithm suited for stream processing. So we take a multiply between C_f and a column of X as example to describe the mapping method.The input stream of transform kernel consists of 4*4 matrix blocks. It brings half waste of APA resources because the computation of four APAs is redundant. Instead, we use MIMD mode of MASA to assign these four APAs to do scalar quantization. In quantization kernel, MF is loaded by look-up table and multiplies Y by index.In SIMD mode on Imagine, we obtain the percentage of efficient operations in adders up to 38%, excluding redundant computation in the other four APAs. The adders in the transform part are executed as butterfly algorithm presents and ones in the quantization part are used for bit-wise operations.

Compared with Imagine, MASA has higher utilization shown in Table 1, because MASA can eliminate redundant computation and provide symmetric AP to loosen the

pressure of adder or multiplier. Thus, IPC of MASA is larger than that of Imagine in SIMD mode. In MIMD mode on MASA, The intermediate results between transform and quantization are stored in scratchpad. As a result, the utilization of AP will increase and the latency of stream load and store will decrease.

Table 1. Utilization of computational unit

Imagine SIMD		MASA SIMD	MASA MIMD
Adder util	Mult util	AP util	AP util
38%	13%	44%	62%

4.3 Entropy Coding

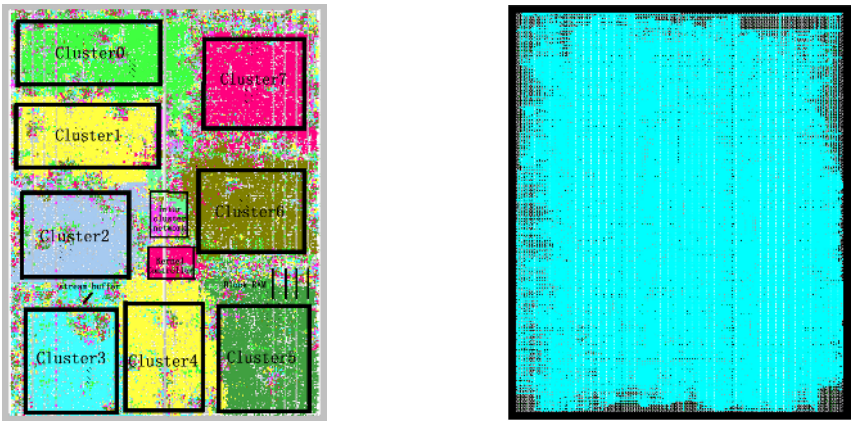
Context-adaptive variable length coding (CAVLC) is a kind of entropy coding method in H.264 encoder. We implement it only in SIMD mode. The total executing time of this kernel is 7352 cycles. The main operations include looking up tables, shifting and patching up the bitstream. All look-up tables are stored in scratchpad in each APA. However, scratchpad bandwidth may limit the main-loop performance in CAVLC kernel.

On our MASA simulator(400Mhz), H.264 encoder can achieve 479 frames per second for a 24-bit 360*288 image (Miss American). The result shows that our H.264 encoder on MASA can process more than 400 frames per second by virtue of multiple models support, exceeding the basic requir ement of real-time coding. Note that MASA is compatible with Imagine in terms of SIMD. As a result, Mpeg standard also can be supported very well in MASA [6]. In conclusion, MASA is flexible enough to implement multiple coding standards.

5 Implementation on FPGA

In order to speedup the emulation and demonstrate the practicability of the proposed framework, we have implemented the prototype MASA architecture on FPGA, which consists of three main top modules: kernel execution module, stream controller and off-chip memory interface. The design is large, to simplify constrains of implementation the design has been partitioned into two FPAGs: a Xilinx XC4VLX200 (2×10^6 logic cells) which includes the SRF and kernel execution module, and a Xilinx XC2VP50 (5×10^6 'gates') which includes stream controller, off-chip memory interface and a PowerPC 405 RISC Core [16].

The architecture was modeled using Verilog as hardware description language. They ware then synthesized using appropriate time constrains with the *Synplify pro* and implemented using *Xilinx ISE*. Figure 6 show the implement view of the MASA on XC4VLX200 which implemented 8 APAs that include totally 48 ALUs. They are fast implemented ALUs by on-chip *XtremeDSP slices* [17]. The SRF and reorder cache are implemented by single port block RAMs in FPGA, while the LRF and kernel instruction store are implemented by Dual Port RAMs. Table 2 summarizes the logic utilization of XC4VLX200 and Table 3 shows the number of resource occupied by each main modules.



Floorplan view Post-place & route view

Fig. 6. Implement views on Xilinx XC4VLX200

MASA must interface with several different types of I/O each running at different clock speeds. For example, the memory controller portion each MBANK must runs at the SDRAM clock speed, but the ALU in APA can run at much higher frequency. Therefore, multiple clock domains are used in MASA's implementation by *digital clock manager blocks* [12] on FPGA. At last, the critical path delay of the implementation of kernel execution module is 7.7 nsec, which assures 130MHz operation frequency. Stream controller and memory interface runs at the half of this clock that is 65 MHz.

Table 2. The logic utilization of XC4VLX200

Logic Utilization	Used	Available	Utilization
Total Number Slice registers	42037	178176	21%
Number of occupied Slices	73616	89088	82%
Number of bonded IOBs	599	960	62%
BUFG/BUFGCTRLs	4	32	13%
Number of RAM16s	232	336	69%
Number of DSP48s	64	96	67%
Number used as logic	107443	-	-
Number used as a rout-thru	40	-	-
Number used as Dual Port RAMs	9444	-	-
Number used as shift registers	17	-	-
Total Number 4 input LUTs	116944	178176	65%

Table 3. FPGA resource occupied by main modules

	#BLKRAM	#DPRAM	#FG	#CY	#DFF	#DSP48
A APA Total of the 8 APAs	1 8	576 4608	6121 48968	144 1152	2948 23584	8 64
Inter-APA network	-	-	483	-	-	-
Kernel controller	72	92	660	60	554	
SRF	64	-	366	-	145	-
Reorder Cache	64	-	321	20	120	-
Stream buffer Total of 16 SBs	- -	- -	2453 19624	55 440	2238 17704	- -
ControlRF&glue logic	24	42	3124	-	50	-
total	232	4742	73546	1672	42057	64

6 Conclusions and Future Work

In this paper, we propose a programmable processor that provides multiple stream execution models and implement key parts for an h.264 encoder on the MASA simulator to investigate its performance. The MASA allows us to explore different parallelism including DLP in SIMD model, DLP & TLP in MIMD model and a hybrid parallelism between DLP and TLP in SKMD model for classes of scenarios. We design kernels and functional units with high functional unit utilization for intensive computation such as block search; DCT and we are investigating ways to minimize the memory references between the computations. Results demonstrate that the processor achieves high performance in h.264 video encoder. Due to compatibility, Mpeg standard also can be supported very well in MASA.

While MASA offers both high performance and flexibility that many media processing applications require, we expect it to replace ASICs in the most demanding of media-processing applications. Our future work is to tune and evaluate the MASA architecture for additional applications and complete the custom design. Another challenge is to create an automatic partitioning and mapping tool assist the user, since kernel partition and stream organization are rather difficult in complex applications.

Acknowledgements. We thank High performance group of Computer School in National University of Defense Technology for helpful discussions and comments on this work. We also thank Stanford Imagine project for providing the simulator of imagine and Xiang Zhong for providing necessary application support. This research was supported by National Natural Science Foundation of China, No.60473080.

Reference

1. http://www.us.playstation.com
2. Jung Ho Ahn et al., Evaluating the Imagine stream architecture, ISCA2004.
3. K. Diefendorff and P. Dubey, "How multimedia Workloads Will Change Processor Design", IEEE computer, 30(9):43-45,Sept.1997

4. Sebastian Wallner, “a Configurable System-on-Chip Architecture for Embedded Devices”, Ninth Asia-Pacific computer system Architecture Conference, BeiJing, Sept.2004
5. T. M. Conte, P. K. Dubey, M. D. Jennings, R. B. Lee, A. Peleg, S. rathnam, M. S. Schlansker, P. Song, A. Wolf, “Challenges to Combining General-Purpose and Multimedia Processors”, IEEE Computrt, pp. 33-37, Dec.1997
6. B.khailany, W.J.Dally, U. J. Kapasi, Peter Mattson, Jinyung Namkoong, J. D. Owens, Brian Towles, Andrew Chang, Scott Rixner, “Imagine: media processing with streams”, IEEE micro, 2001.3/4
7. Mei Wen, Nan Wu, Haiyan Li, Li Li, Chunyuan Zhang, “Multiple-Morphs Adaptive Stream Architecture”, Journal of Computer Science and Technology, 2005.9
8. Nuwan Jayasena, Mattan Erez, Jung Ho Ahn, and William J. Dally, “Stream Register Files with Indexed Access”, Tenth International Symposium on High Performance Computer Architecture, Madrid, Spain, February 2004.
9. 2003 Workshop on Streaming Systems,http://catfish.csail.mit.edu/wss03/.
10. M. B. Taylor et al., “The Raw Microprocessor: A Computational Fabric for Software Circuits and General Purpose Programs”, IEEE Micro, 2002 ¾.
11. 11.Karthikeyan Sankaralingam et al., “Exploiting ILP, TLP, and DLP with the Polymorphous TRIPS architecture”, 30th Annual International Symposium on Computer Architecture, May 2003.
12. http://www.flipcode.com/articles/article_dx8shaders.shtml
13. Mei Wen, Nan Wu et al., “A Parallel Reed-solomon Decoder on the Imagine Stream Processor”, Second International symposium on Parallel and Distributed Processing and Applications, LNCS3358, Hongkong, 2004.12
14. Mattson, “A Programming System for the Imagine Media Processor”, Stanford Ph.D. Thesis , 2001
15. Zhibo Chen, Peng Zhou, Yun He, “Fast Integer Pel and Fractional Pel Motion Estimation for JVT”, 6th Meeting: Awaji, December 2002
16. http://www.xilinx.com, “Virtex-II Pro and Virtex-II Pro X Platform FPGAs:Complete Data Sheet”, DS083 (v4.2) March 1, 2005
17. http://www.xilinx.com, “Virtex-4 Family Overview”, DS112 (v1.3) March 26, 2005
18. 18 Haiyan Li, Mei Wen, Nan Wu, Li Li, Chunyuan Zhang, “Accelerated motion estimation of h.264 on Imagine stream processor”, International conference on Image analysis and recognition 2005, LNCS, Toronto, 2005.9

The Challenges of Massive On-Chip Concurrency

Kostas Bousias and Chris Jesshope

Computer Systems Architecture Group
Instituut voor informatica
Universiteit van Amsterdam
{jesshope, bousias}@science.uva.nl

Abstract. Moore's law describes the growth in on-chip transistor density, which doubles every 18 to 24 months and looks set to continue for at least a decade and possibly longer. This growth poses major problems (and provides opportunities) for computer architecture in this time frame. The problems arise from current architectural approaches, which do not scale well and have used clock speed rather than concurrency to increase performance. This, in turn, causes excessive power dissipation and circuit complexity. This paper takes a long-range position on the future of chip multiprocessors, both from the micro-architecture perspective, as well as from a systems perspective. Concurrency will come from many levels, with instruction and loop-level concurrency managed at the micro-architecture and higher levels by the system. Chip-level multiprocessors exploiting massive concurrency we term *Microgrids.* The directions proposed in this paper provide micro-architectural concurrency with full forward compatibility over orders of magnitude of scaling and also the management of on-chip resources (processors etc.) so as to autonomously configure a system for a variety of goals (e.g. low power, high performance, etc.).

1 Introduction

1.1 Micro-architecture Challenges

Although today's large scale parallel computing systems comprise clusters of commodity processors, discs and networks, future systems will have to address fundamentally new issues as we inevitably move towards large-scale, on-chip parallelism, i.e. from 10^3 to 10^5 processors, which we call *Microgrids*. Microgrids will also form the basis of mega-scale computing systems, comprising millions of processors, compounding the issues and increasing system-management complexity. To fully exploit such complex systems it is essential to answer some fundamental questions that simplify and give a formal basis for on-chip concurrency models, execution strategies and resource management. Failure to address these problems has delayed the introduction of highly concurrent micro-architecture [1] and consequently, technology advances over the last decade have advanced processor performance through clock speed, using a combination of smaller gate delays and shorter pipeline stages. Performance gains from concurrency have been limited, even though circuit density has grown more rapidly than circuit speed. Instead, increased circuit density is supporting unscalable execution models, such as out-of-order issue (OoO). Ironically, large on-chip memories are then used to mitigate against the divergence between on-chip clock speeds and

T. Srikanthan et al. (Eds.): ACSAC 2005, LNCS 3740, pp. 157 – 170, 2005.

memory cycle times, resulting from the aggressive clocking. OoO has a long history going back to Tomasulo's algorithm introduced in the IBM 360/91 [2] and subsequent developments such as reorder buffers [3], which are used extensively in modern microprocessors. However, instruction dispatch [4] and register file [5] implementations have poor scaling properties and this has led to several, recent, high-profile projects being cancelled due to excessive circuit complexity and power dissipation e.g. [6]. This power barrier should have been no surprise, as in 1999 it was predicted that the Alpha 21464 would use a quarter of the chip's power budget on its instruction queue [7]; this chip was also cancelled in 2002.

Using concurrency to obtain performance is a much better strategy, as long as some fundamental questions can be answered. To illustrate this consider the T800 transputer [8], a 0.25M transistor chip with 64-bit floating point capability, designed for concurrent applications. This processor could be replicated 400 times in current technology, giving instruction issue widths a hundred times those found in current OoO processors. Such naïve chip multi-processors (CMPs) are not particularly viable in a general-purpose market, as they require explicit, user-level, concurrency to program them making the migration to such systems slow and difficult. The one and only advantage of the OoO paradigm is that concurrency is extracted and exploited implicitly from legacy binary code. Very-long Instruction words (VLIW) are used increasingly in embedded applications and also have problems in scaling up to massive concurrency. Here scheduling is delegated to the compiler, which although produces lower-power solutions to instruction issue, also generates static schedules that limit scalability. The first challenge we face therefore, is to obtain scalability across a wide range of codes while retaining binary- and source-code backward compatibility.

In [9,10] a number of technological challenges are outlined for future micro-architectures. These include scalability of micro-architectures in area, performance and power dissipation, as well as strategies for chip multiprocessors that address power awareness and power management. Finally, there is the issue of signal propagation, which will force micro-architectures to eliminate global on-chip communication completely. One of the major global communication networks is the clock-distribution network and a more practical approach to future CMP design would be to use a globally-asynchronous, locally-synchronous (GALS) clocking approach but the big question is how to design ILP processors, which naturally synchronise on register variables, with an asynchronous and distributed model of communication. This is the challenge undertaken in this paper.

1.2 Micro-architecture Concurrency

There are two widely-used models of concurrency at the micro-architectural level. The first is implicit and relies on hardware to detect and enforce dependencies when executing instructions out of order. As already indicated this model scales very badly when increasing concurrency. The other model is VLIW, which has better scaling but has compatibility problems. In particular, binary code must be regenerated (as the schedules are static) for each increase in concurrency. EPIC architectures provide some remission in this area by allowing the binding of instruction to resource to be dynamic. However, it requires many of the structures found in OoO approaches, such as branch predictors and the static schedules limit scalability.

A third way has been explored by a number of groups; it relies on decomposing and managing multiple fragments of code concurrently. The scheduling of these code fragments must be made efficient and this requires the fragments to be exposed within a single context, which differentiates it from most multi-threaded architectures. By interleaving fragments, latency tolerance is achieved and by distributing fragments to different functional units or processors, speedup is obtained. The first published paper on code fragmentation was called *microthreads* and dates back to 1995 [11]. It was proposed as a means by which processors in a distributed system could tolerate high levels of latency. More recently a similar approach called *intrathreads* [12] adopts the same principal but with a different approach to implementation. It uses bounded concurrency and statically-partitioned resources, whereas microthreads describe parametric concurrency where resources are managed dynamically though the concept of micro-contexts. Another difference is that intrathreads separate synchronisation and data storage, where microthreaded processors implement registers as i-structures synchronise between code fragments. In recent papers, microthreading has been extended to support CMPs [13,14] and simulated to evaluate latency tolerance and speed up [15,16].

These models are both incremental and add just a few new instructions to an existing ISA to implement explicit concurrency controls. In microthreading, these instructions define parametric sets of concurrent code fragments, which are scheduled dynamically on multiple processors. In intrathreads, the number of threads is fixed and the implementation is targeted to wide-issue pipelines rather than to a chip multi-processor. A key feature of microthreads is that concurrency is parametric but that schedules are dynamic. The same binary code can therefore be run on an arbitrary number of processors, limited only by the parametric concurrency. This allows for the dynamic management of resources in microgrids. Thus the number of processors can be set dynamically to satisfy constraints on performance or power dissipation without modifying the binary code. It will be demonstrated in this paper that a number of tradeoffs can provide management of power and performance over an extremely wide range of processors using largely linear functions. This makes the model ideal for autonomous configuration.

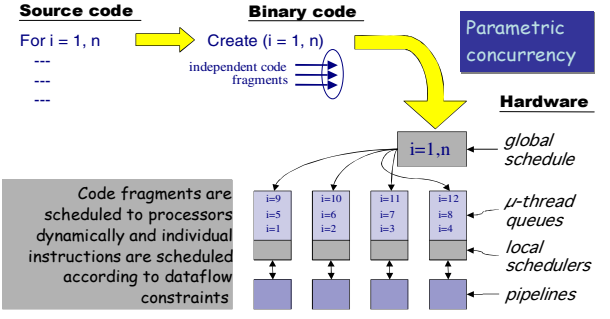


Fig. 1. An illustration of the concept of microthreaded scheduling

Microthreading is applied within a single context and supports a shared-register model of data using blocking reads (code fragments are suspended on registers waiting for data to be written). Memory consistency is managed using a barrier synchronisation instruction, which forces single-threaded, in-order execution. It has been shown [1,17] that implementations of this model are completely scalable and support asynchronous inter-processor communication that is tolerant to communication delay and does not force any pipeline stalls. This approach therefore, solves many of the challenges raised in [9,10]. This

model can exploit the full benefit of scaling due to Moore's law to the end of silicon (i.e. an estimated 10^3 to 10^5 processors per chip).

Microthreaded concurrency is obtained by compiling source code using a microthread-aware compiler (or by a translation from existing binary code). The model requires in-order execution semantics, which means that optimal static schedules can be generated for instruction sequences and deterministic compiler optimisations can be applied. The exception is where data generation is non-deterministic, as in inter-thread communication, cache access or iterative operations. When attempting to read such data, an explicit context switch is signalled to the hardware and another code fragment is executed. Concurrency is captured by instructions that create families of threads to execute loops for all values of their index variable concurrently; basic-block concurrency can also be captured. The former is parametric and the latter is static. There are constraints on the creation of code-fragment instances due to resource availability and also dataflow constraints on the execution of individual instructions. These constraints determine the dynamic schedule for an execution of the code on a given number of processors.

1.3 Resource Management

Resource management is currently divided between two very different domains. At the processor level, it assumes that computation is performed using a single, powerful processor and a large global memory system. The memory holds images of all current activities, i.e. operating system and user tasks. Resource management is then almost entirely undertaken by sharing processor cycles between these tasks. The large memory is required to store the state of the multiple tasks and is slow. A memory hierarchy is then used to solve this problem by caching the data close to the processor, using implicit data transfers. This is not an optimal solution for several reasons:

- ideally, memory should be distributed to make it faster;
- transfers to cache can be initiated early to achieve tolerance to the high latency memory, interaction between tasks can interfere with implicit and explicit transfers making optimal solutions heuristic rather than deterministic;
- finally, moving state between levels of memory will aggravate bandwidth requirements and cause significant power dissipation.

Solutions to these problems can be found in recent research on processing in memory architectures (PIM) [18] and, as this paper will show, using microthreading combined with novel resource management solutions, while using near-conventional instruction sets with full code compatibility.

The second resource-management domain is on the scale of meta-computers in Grid infrastructures. This assumes the low-level resource management described above and provides on top of that, some measure of service quality by resource reservation and application adaptation. This is usually implemented as a middleware layer on top of one or more conventional operating systems, (e.g. [19,20]). Neither domain provides solutions to resource management at the micro-architecture level when dealing with large numbers of processors. A middleware solution is too coarse grain and conventional operating systems are more suited to single processor. environments.

Recently, much emphasis is being placed on the optimisation of power dissipation. Attempts have been made to manage power dissipation as a function of issue width in

speculative processors [21-24]. These solutions rely on code profiling but dynamic concurrency varies significantly leading the use of dynamic hardware profiling [25]., which increases power usage and so there are limits on the scope of such techniques. Ideally, compiler-based solutions are preferred, e.g. [26] but such approaches can only effectively control the cache power-performance. Combined compiler and hardware approaches using fetch throttling can also control concurrency [27] but give only marginal impact, i.e. 10 to 20% power savings with similar performance degradation through lower IPC. A second and profound question then is how to design systems of thousands of processors managed from legacy code, while optimising various goals such as performance, power and responsiveness over orders of magnitude?

2 Microgrids

2.1 New Processor Architectures

Microthreading provides parametric concurrency using a few additional instructions to manage fragments of code efficiently. It adopts a shared-register model of data with synchronisation on all registers. Assume *ISA-mt* are the additional microthreaded instruction, then given a RISC processor whose instructions are defined by *ISA-RISC*, then a new ISA can be defined incrementally by the union of the two, i.e. ISA-RISC + ISA-mt. Similarly we could define a VLIW microthreaded architecture by the union of a different instruction set, ISA-VLIW with ISA-mt. This paper is concerned only with the issues arising from ISA-mt.

ISA-μt has been fully specified in [1]. Using that definition, families of code fragments can be specified using the following two concepts:

i. *Sets* of code fragments statically define concurrency within a shared register domain. Each code fragment is specified by a pointer to its first instruction $\{P_i, 0 \le i \le n-1\}$ and is terminated by a *Kill* instruction. There is no restriction on communication between sets of code fragments.
ii. *Iterators* dynamically define concurrency over a set of code fragments. An iterator specifies an integer index variable, i, using a triple {s,l,t} such that $\{i \ge s; i \le l; i = s+kt\}$ where k is a positive integer. The code defined by the set is shared between iterations by defining a micro-context for each value of i. A micro-context is a partition of a processor's physical register space allocated to an iteration. The first location of the micro-context is initialised to the index value, i. In the current model, communication between micro-contexts is restricted and an iteration can access the micro-context of just one other prior iteration, defined by a constant stride in the index space, d. The number of registers in the micro-context and the value of the stride complete the definition of an iterator.

Less restrictive models can also be defined to increase the potential concurrency exposed, for example in allowing multiple, constant-strided dependencies or even variable-strided dependencies. However, as concurrency is parametric and unrelated to machine resources, these models may induce resource deadlock, which is not easily

resolved. This paper considers only the simplest model above, where conditions under which resource deadlock occur can be easily specified.

2.2 New Chip Architectures

A *Microgrid* is a chip comprising, in our model, M microthreaded processors, where each processor (or cluster of processors) is implemented in its own, clocked domain and communicates asynchronously with the rest of the chip. Each processor has a local register file, access to a broadcast bus and a ring network for shared-register communications between neighbouring processors, see [1] and figure 2. When executing a single thread of control (a *Context*), a number of processors can be used to execute an iterator, this is called the *Profile* associated with that context. A profile is an ordered subset of processors of cardinality P selected somehow from the M processors in the microgrid. This subset is configured to have a broadcast bus and ring network linking only the processors within it.

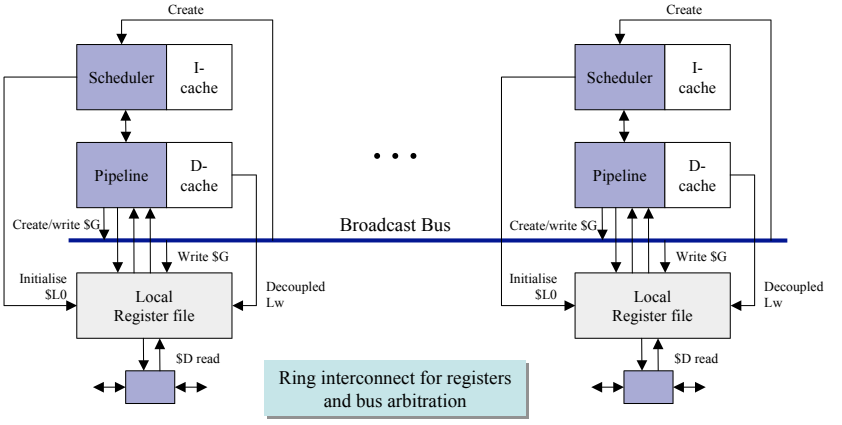


Fig. 2. A chip multi-processor based on a collection of microthreaded pipelines. The ring network and broadcast bus (which may be implemented over the ring) provide asynchronous communication between processors decoupled from the local pipeline operation.

During the execution of an iterator, each processor's register file will contain the state for the current context plus micro-contexts for iterations scheduled to it. For a given P, the triple defining an iterator and a processor's position in the profile, it can independently determine the iterations it must execute. These are initialised when it has resources available, in the form of handles to identify its code fragments and registers to allocate to its micro-context. The distribution of iterations to processors is defined by a simple modulo mapping such that iteration `i: s≤i≤l`, is allocated to processor `j: 0≤j< P`, where:

$$j = \left| i/kd \right|_{P} \qquad (2.1)$$

Thus blocks of `kd` consecutive iterations are allocated to each processor in turn, where `d` is the communication stride and `k` is a locality parameter, a natural number limited by the number of registers in the local register file, `R`. If the static context re-

quires G registers and each micro-context requires L registers, then to avoid resource deadlock the following inequality must be satisfied:

$$kdL + G \leq R$$

Normally k should be maximised subject to the above constraint, as k-1 is the ratio of local to remote communications in a loop-carried dependency chain. Note that k also determines locality in the local D-cache if implemented and can be chosen to take full advantage of the line size, data size and access patterns to local data. When executing independent loops, there are no issues with resource deadlock and an arbitrary allocation of iterations to processors is possible as there is no communication between micro-contexts.

2.3 Power Awareness and Low-Power Operation

The final issue to be considered before a microgrid operating environment can be discussed is that of power models for microthreaded microprocessors. As we have already seen in [21-27], managing power using implicit concurrency is difficult and has limited effectiveness. Implicit ILP relies on speculation and eager instruction execution policies, which are at odds with power conservation. Microthreaded microprocessors, on the other hand, have conservative instruction execution policies that enable power-aware operation and yield power-efficiency. Only a single instruction per code fragment is fetched at a time and branch and data hazards suspend execution of that fragment until the hazard has been resolved; execution can continue from other fragments if any are active. In the case of a branch hazard the fragment is suspended for a few cycles until the branch-target address is computed but with a data hazard the code fragment is suspended indefinitely on a register until the required data has been written (if the data already exists execution is suspended only long enough to discover that fact). Conservative instruction issue policies enhance power efficiency, as:

- no power is dissipated on speculative instruction fetch and execution;
- no area and power are required in making branch or data predictions;
- no area and power are required in managing missprediction cleanup;
- finally, conservative models provides signals to manage power dissipation.

Code fragments are managed by a scheduler, which selects a new code fragment for execution on any instruction from ISA-μt that causes a context switch. It also manages the state of all allocated fragments, i.e. whether they are active or suspended. When all fragments are suspended, this could be used to trigger a higher level context switch but as already noted, this will involve significant data movement and power consumption. Alternatively this state can be actively used to manage local power dissipation. Each processor runs its own clock and that clock can be stopped awaiting data, eliminating any dynamic power dissipation. By definition, as no local threads are active, the wake-up signal must arrive externally, either from the bus, the ring network or from memory. These inputs are managed by an asynchronous interface to the external read/write port of the register file, which can signal the scheduler to restart the local clock when it re-schedules the suspended fragment. The same signal can be used to manage a processor's power rails and minimise static power dissipation when idle. Note that such control is not possible in speculative execution policies.

This policy allows us to statically place single contexts on dedicated processors, which become idle while waiting for external events. This distributes the locus of control of many tasks and localizes the use of memory, avoiding excessive and unnecessary migration data between different levels of memory. If further, the computation can be described at a higher level as a communicating collection of components (i.e. a streaming network), the tasks can be data driven and the only movement of data will be due to explicit algorithmic concerns, rather than interference in cache memory from scheduling all tasks to a single monolithic processor.

3 Microgrid System Environment

It is first useful to summarise the properties of microthreaded pipelines before looking at how a system environment can be implemented to manage the processors in a microgrid. The properties relevant to this discussion are that:

- microthreaded binary code captures parametric concurrency and those parameters can be set dynamically;
- microthreaded binary code is schedule invariant and can be executed unchanged on a number of processors up to a limit defined by the parameters;
- instructions are tolerant of high levels of latency in their operands;
- processors have asynchronous interfaces and are independently clocked;
- processors consume minimal power while waiting for external events.

An operating system environment to support the massive on-chip concurrency proposed requires new paradigms to be adopted. Microthreaded ISA extensions allow concurrency to be extracted from legacy code through binary code translation or recompilation. Because this code is schedule-independent the environment can support the allocation of dynamic profiles to contexts. There is also a need to execute unmodified binary code from the base ISA on a single processor. Thus the system environment must provide support in launching contexts and in adjusting their profiles. This would support all forms of concurrency such as user jobs, multithreaded applications etc. and be flexible enough to support explicitly programmed concurrent applications in new programming paradigms, e.g. [28,29]. A strategy is proposed below for building such a System Environment Process (SEP). It assumes:

- there are a large number of processors on a chip (e.g. 10^3 - 10^5);
- contexts are allocated to processors for their duration;
- contexts communicate using shared memory and/or I/O, managed by microthreads and microcontexts running on dedicated processors;
- one processor runs a “kernel” (the SEP), that manages a model of the system resources and is responsible for launching contexts and configuring profiles.

The profile for a microthreaded context is defined as the number of processors allocated to it, at a given time. A profile is initially a single processor, when the context is launched. Later, if the context exploits ISA-μt, then more then one processor can be used to execute the created code fragments. The processors are added to the context dynamically by requests to the SEP. The choice of profile can be used to optimise the chip’s performance according to goals and environmental factors. The times at which a profile may change are during single-(micro)threaded execution, i.e. following a barrier synchronisation and prior to the next create instruction [1]. At these times no micro-contexts exist and the context is fully defined by its state on a single processor.

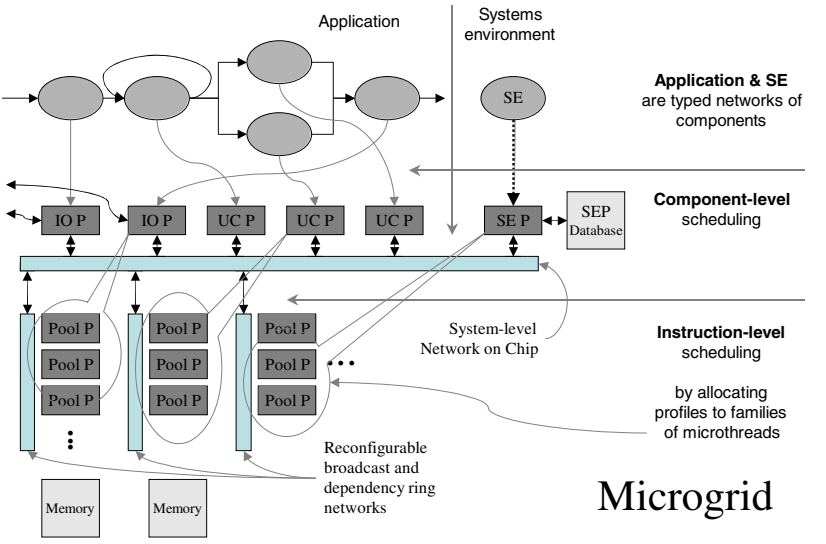


Fig. 3. The microgrid concept showing various levels of scheduling for the on-chip resources. IOP = I/O processor; UCP = user context processor, SEP = system environment process.

The simplest form of profile is where a context is allocated a fixed number of secondary processors for the duration of its execution; a more dynamic profile might allocate resources at function boundaries and the most dynamic would be adjusted at the level of individual create instructions (loops). Thus the time constant for reconfiguration would varies from human interaction speeds down to once every 1000 or so processor cycles, i.e. from O(1) to O(10^6) times a second, spanning at least 6 orders of magnitude. The lower estimate is based on the number of registers in a processor assuming that each is written at least once during the optimal execution of a loop.

In practice, the smallest time constant in setting a profile will depend on a number of issues. These include the characteristics of the code, the requirements for its execution (e.g. minimum time, maximum throughput, maximum latency tolerance, minimum power etc.) and the time required to configure the profile following a request to the SEP. The latter involves several transactions on an in-memory database and the configuration of a ring network. Only after the configuration is complete can a Create instruction broadcast a pointer to its parameters to the new profile. Then, each processor autonomously executes its schedule as defined by equation 3.1.

Requests for a profile must be embedded in the compiled code at compilation or binary translation and are remote requests to the SEP, where a global model of all resources is maintained. The SEP also manages the configuration of ring networks. It is important to understand that an execution of the code is unaffected by the number of processors used, except in terms of speed and power dissipated. It is even possible to loosely couple the allocation process and the execution of a subsequent family of microthreads. The simplest protocol would involve:

i. a context making an SEP request;
ii. the SEP updating its profile model, configuring a new set of resources but not binding them to the context's current environment;
iii. both SEP and context competing on the system bus and either:
    a. the context winning, executing a create to the old profile and releasing the bus – the SEP would re-absorb the unused resources;
    b. the SEP winning, binding the new resources to the old and releasing the bus – the context requires this to execute the create.

Resource management therefore uses a client-server model and exploits the schedule independence of microthreaded code. Using the above protocol, requests for resources are non-binding and non-blocking and are a part of the compiled code. Non-microthreaded code gets a single processor and benefits from power-awareness without using microthreaded instructions or dynamic profiles.

4 Enabling Results

In order to demonstrate the feasibility of this approach, results are presented that demonstrate the scaling characteristics necessary for it's operation. They are based on a simulation of a CMP shown in figure 2, using a seven-stage Alpha pipeline, with parameters as defined in table 1. Each processor executes instruction in-order without branch prediction. The CMP is used to evaluate the performance and power scaling assuming fine-grain regulation of dynamic power in the scheduler as described in section 2.3. The code executed is a hand-compiled fragment that implements the Livermoore hydro fragment. The higher-level microgrid architecture is assumed to be ideal, with a relatively slow but non-blocking second level of shared memory. This is a reasonable assumption for regular computation as data can be partitioned according to the a-priori schedules. Scheduling information can also be used to optimize the L1 D-cache hit rate. The scope for optimisation in microgrids, which have simple, regular schedules, is much higher than in a modern superscalar processor, where all aspects of code execution are speculative and heuristic.

The L1 cache controller must quash multiple requests to memory for the same cache line. Association between address and target register is managed by tagging requests with the register specifier and can be buffered anywhere in the memory system. Note that with regular schedules and an 80% cache hit rate, only 2-3% of memory loads cause a request to the second-level memory, which is pipelined and provides a line of 64 bytes in 24cycles. Transfer is in 8-byte words and the requested word is returned first in 10 cycles.

The results presented in Figure 4 which shows the speedup for the Livermore hydro fragment over a wide range of profile sizes executing the same binary code. The simulation is performed with cold caches and includes the overhead of thread creation and barrier synchronization following the execution of the loop. The schedule maps 16 consecutive iterations per processor, in order to optimise the L1 D-cache hit rate. The speedup is within 3% of the ideal for up to 256 processors and is still within 20% of the ideal for a profile of 2048 processors. The loss of efficiency for larger profiles is due to the amortisation of start-up overheads over fewer cycles and less latency tolerance resulting from the fixed problem size of 64K iterations. This gives only 32

iterations per processor for the largest profile or a 20% utilisation of the processors' resources. Figure 4 also shows speedup against non-microthreaded, single-processor execution and here the speedup is super-linear for all profiles. Note that microthreaded code executes about 20% less instructions than non-microthreaded code to achieve the same result, as index and loop control operations are "executed" in the scheduler. Management of control and data hazards also contributes to the superior single-processor performance of microthreaded code, although but other architectures will have different solutions to this that are not simulated here. Nevertheless the single processor profile achieves an IPC of 99.8% on this code even in the presence of cache misses and a slow second-level memory.

Table 1. Parameters for simulations

System parameters

Main memory Size	4 MB
Local registers / processor	1024
LCQ entries / processor	512

I-cache parameters

Line Size	32bytes
Associativity	8
No Of Sets	8
Buffer entries	2

D-cache parameters

Line Size	64bytes
Associativity	8
No Of Sets	128
Buffer entries	512

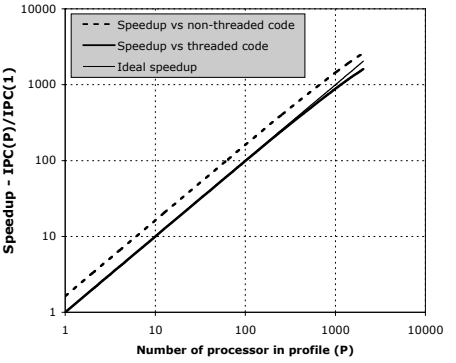


Fig. 4. Speedup of hydro-fragment against number of processors in microgrid profile

The simulator also models power dissipated and figure 5 shows the results, which assume the processors' clocks are enabled on a cycle by cycle basis depending on whether the scheduler has active threads to execute or not. It also assumes that all processors in a profile dissipate static power for the duration of the computation. The results are presented as the relative energy consumed by the hydro-fragment kernel as a function of the number of processors in its profile. The total energy assumes that a processor consumes the same amount of

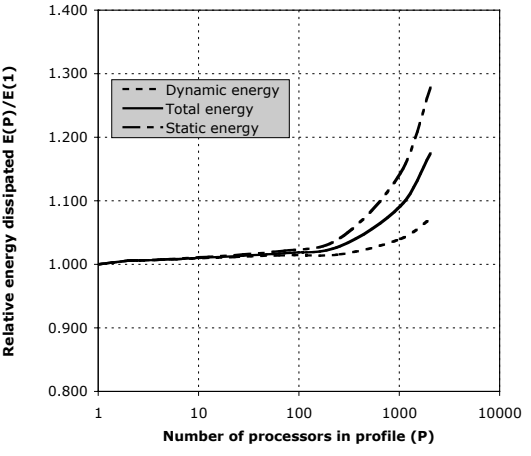


Fig. 5. Relative energy dissipated for the same computation using different numbers of processors in a profile

energy from dynamic power and static power dissipation. These results also assume that processors not in a profile consume no power.

Regardless of the balance between static and dynamic power, the envelope between static and dynamic energy in figure 4 will contain the total energy. Again it can be seen that over two and a half orders of magnitude the power dissipations remains within 3% of that consumed by executing the computation on a single processor. At 2048 processors the efficiency of the computation decreases due to inefficient use of the pipeline and a 17.5% overhead in energy consumed is seen. This is a very significant result as it opens up a mechanism for very low-power computation. Figure 4 shows power dissipated for scaled performance using a large number of processors. When using multiple processors for constant performance, their frequency can be reduced by a factor equal to the number of processors used. Scaling the voltage with frequency would reduce the total energy required by a factor close to the square of the processors used!

5 Conclusions

In this paper the microthreaded model of concurrency has been summarised and the concept of a microgrid, based on massively concurrent and asynchronous collections of microthreaded processors has been proposed. It is argued that microgrids need new concepts of operating environments and that this approach can exploit Moore's law to the end of silicon. The proposed approach exploits massive concurrency by statically placing user and systems contexts rather than time-sharing them. This is combined with dynamic profiles for contexts that use microthreaded instructions. The schedulers in each processor can exploit the model's parametric concurrency by autonomously organising the computation across all processors in the profile. Higher-level control of this mechanism can be used to optimise a number of system goals over a wide range of parameters. The results presented show a linear performance scaling on independent loops that spans profiles ranging over two and a half orders of magnitude for a fixed sized problem. The performance over this range deviates from ideal scaling by only 2% and the energy required does not grow by more than 3% despite the additional performance.

In a microgrid, different contexts would draw dynamically from a pool of processor resources rather than being time-sliced on a single powerful processor. All contexts thus retain a minimal profile of one processor, even though that processor may be idle for much of the time. With 10^5 processors available, there will no lack of processors. Also many low-frequency (system) tasks can be scheduled as collections of microcontexts on a single processor, responding efficiently to external events, such as timers or I/O and not wasting system resources. Even if these root processors are idle, this strategy still makes sense, so long as the idle processors do not consume power. In general, a good systems management strategy must minimise the critical resource usage and in a microgrid, this is not processor cycles! Rather it will be power dissipated, memory bandwidth, chip I/O and perhaps other characteristics. This static allocation of minimal profile plus dynamic allocation of additional processors can be based entirely on compiled user-code and most importantly the requests for resources are both non-binding and non-blocking.

There are many research questions still to be answered in providing a foundation of tools and interfaces for the control and optimisation of these critical resources but this paper demonstrates the scalability that underpins this approach. Clearly it introduces massive design space optimisation issues, however, as the tradeoffs are largely linear, the exploration can be simplified and even made dynamic and embedded into the systems environment model.

References

1. 1.C Jesshope (2004) Micro-grids - the exploitation of massive on-chip concurrency, invited paper, Cetraro HPC workshop 2004, Cetraro, Italy, to be published in *Advances in Parallel Computing*, 2005 (http://staff.science.uva.nl/~jesshope/Papers/HPC-paper.pdf)
2. D W Anderson, F J Sparacio, R M Tomasulo (1967) The IBM System/360 Model 91: Machine Philosophy and Instruction-Handling, *IBM J. Res. Dev.*, **11**, Number 1, p8.
3. J Smith and A Pleszkun (1985) Implementation of Precise Interrupts in Pipelined Processors, in *Proc. of Int'l. Symposium on Computer Architecture*, pp.36—44.
4. 4.G Kucuk, D Ponomarev, K Ghose and P M Kogge (2001) Energy--Efficient Instruction Dispatch Buffer Design, in *Int'l. Symp. on Low Power Electronics and Design* (ISLPED'01), August 2001.
5. R Balasubramonian, S Dwarkadas and D Albonesi (2001) Reducing the Complexity of the Register File in Dynamic Superscalar Processor, in *Proc. of the 34th Int'l. Symposium on Microarchitecture* (MICRO'01), p37.
6. 6.A Shilov (2004) Intel to Cancel NetBurst, Pentium 4, Xeon Evolution, http://www.xbitlabs.com/news/cpu/display/20040507000306.html, accessed 7/1/2005.
7. 7.K Wilcox and S Manne (1999) Alpha processors: A history of power issues and a look to the future, Cool-Chips Tutorial, November 1999. Held in conjunction with MICRO-32.
8. 8.M Homewood, D May, D Shepherd and R Shepherd (1987) The IMS T800 Transputer IEEE Micro, October 1987, pp10-26.
9. 9.R H J M Otten and P Stravers (2000) Challenges in physical chip design, *Proc. ICCAD '00*, p84, San Jose, California, IEEE Computer society press.
10. R Ronen, A Mendelson, K Lai, F Pollack and J Shen (2001) Coming challenges in Microarchitecture and architecture. Proc IEEE, 89 (3), pp325-40.
11. A Bolychevsky, C R Jesshope and V B Muchnick, (1996) Dynamic scheduling in RISC architectures, *IEE Trans. E, Computers and Digital Techniques* , **143**, pp309-317
12. A Gontmakher and A Schuster (2002) Intrathreads: Techniques for Parallelizing Sequential Code, *6th Workshop on Multithreaded Execution, Architecture, and Compilation (MTEAC-6)*, November 2002, Istanbul (in conjunction with Micro-35).
13. C. R. Jesshope and B. Luo (2000) Micro-threading: A New Approach to Future RISC, *Proc ACAC 2000,* pp34-41, ISBN 0-7695-0512-0 (IEEE Computer Society press), Canberra Jan 2000.
14. Jesshope, C. R. (2001) Implementing an efficient vector instruction set in a chip multiprocessor using micro-threaded pipelines, Proc. ACSAC 2001*, Australia Computer Science Communications*, **23**, No 4., pp80-88, IEEE Computer Society (Los Alimitos, CA, USA), ISBN 0-7695-0954-1.
15. Luo B. and Jesshope C. (2002) Performance of a Micro-threaded Pipeline, *in Proc. 7th Asia-Pacific conference on Computer systems architecture*, **6** , (Feipei Lai and John Morris Eds.) Australian Computer Society, Inc. Darlinghurst, Australia, ISBN ~ ISSN:1445-1336 , 0-909925-84-4 , pp83-90.

16. Jesshope C. R. (2003) Multithreaded microprocessors – evolution or revolution (Keynote paper), *Proc. ACSAC 2003: Advances in Computer Systems Architecture*, Omondo and Sedukhin (Eds.), pp 21-45, Springer, LNCS 2823 (Berlin, Germany), ISSN0302-9743, Aizu, Japan, 22-26 Sept 2003.
17. Jesshope C. R. (2004) Scalable Instruction-level Parallelism, *In Computer Systems: Architectures, Modelling and Simulation, Proc 3rd and 4th Int;l. Workshhops, SAMOS 2003, SAMOS 2004*, (LNCS 3133, Springer), ISBN 3-540-22377-0, pp383-392, presented Samos, Greece, July 2004.
18. Brockman J. B., Thoziyoor S., Kuntz S. K. and Kogge, P. M. (2004) A low cost, multithreaded processing-in-memory system, *Proc. 3rd workshop on Memory performance issues* (ISCA 31), ISBN:1-59593-040-X, pp16 – 22.
19. I Foster, A Roy and V Sander (2000) A Quality of Service Architecture that Combines Resource Reservation and Application Adaptation, *Proc. 8th International Workshop on Quality of Service* (IWQOS 2000), Pittsburgh, USA.
20. R Buyya and M Murshed, (2002) GridSim: A Toolkit for the Modeling and Simulation of Distributed Resource Management and Scheduling for Grid Computing, *Journal of Concurrency and Computation*: *Practice and Experience* (CCPE), **14**(13-15), Wiley Press, Nov.-Dec., 2002
21. S Manne, A Klauser, and D Grunwald, (1998) Pipeline Gating: Speculation Control for Energy Reduction, *Proc Intl Symp. On ComputerArchitecture* (ISCA).
22. D Marculescu (2000) Profile driven Code Execution for Low Power Dissipation, *Proc. Intl. Symp. Low Power Electronics and Design*.
23. V Zyuban and P Kogge (2000) Optimization of High-Performance Superscalar Architectures for Energy-Delay Product, *Proc Intl. Symposium on Low Power Electronics and Design*.
24. S Ghiasi, J Casmira and D Grunwald, (2000) Using IPC Variation in Workloads with Externally Specified Rates to Reduce Power Consumption, *Workshop on Complexity Effective Design. ISCA27.*
25. A Iyer and D Marculescu (2001) Power aware microarchitecture resource scaling, Proc Design, automation and test in Europe, Munich, Germany, pp190-6, ISBN:0-7695-0993-2.
26. R Sree, A Settle, I Bratt and D A Connors (2003) Compiler-directed resource management for active code region, *Proc. 7th Workshop on Interaction between Compilers and Computer Architecture*, February 2003, 5.5.
27. S Chheda, O Unsal, I. Koren, C M Krishna and C A Moritz (2004) Combining compiler and runtime IPC predictions to reduce energy in next generation architectures, *Proc. 1st Conf. On Computing Frontiers archive*, Ischia, Italy, pp 240-54 , ISBN:1-58113-741-9.
28. A Shafarenko and S-B Scholz (2005) General Homomorphic overloading. Accepted for publication in LCNS (2005). A preliminary version was submitted to IFL'2004 in Luebeck.
29. A Shafarenko (2003) Stream Processing on the Grid: an Array Stream Transforming Language. SNPD 2003: 268-276.

FMRPU: Design of Fine-Grain Multi-context Reconfigurable Processing Unit

Jih-Ching Chiu and Ren-Bang Lin

Department of Electrical Engineering,
National Sun Yat-Sen University, Kaohsiung, 804, Taiwan
chiujihc@ee.nsysu.edu.tw

Abstract. At present the scale of multimedia and communication systems has become more and more complicated due to their fast developments. In order to handle diverse functions and shorten system development time, the ability to reconfigure system architecture becomes an important and flexible design consideration. In this paper, we propose a novel reconfigurable processing unit, FMRPU, which is a fine-grain with multi-context reconfigurable processing unit targeting at high-throughput and data-parallel applications. It contains 64 reconfigurable logic evel connectivity. According to the simulation results, the longest routing arrays, 16 switch boxes, and connects with each other via three hierarchical-lpath of FMRPU only takes 6.5 ns at 0.35 processes, which is able to construct the required logic circuit efficiently. To compare with same kind devices in dealing with Motion Estimation operations, the performance is raise to 17% and has excellent performance in executing DSP algorithms.

1 Introduction

Nowadays due to the fast development of multimedia and communication applications, reconfigurable computing is becoming an important part of research in computer architectures and software systems. This is because it has potential to greatly accelerate a wide variety of applications. By placing the computationally intense portions of an application onto the reconfigurable hardware, that application can be greatly accelerated. This is because reconfigurable computing combines many benefits of both software and ASIC implementations. Like software, the mapped circuit is flexible, and can be changed over the lifetime of the system or even the lifetime of the application. Similar to an ASIC, reconfigurable systems provide a method to map circuits into hardware. Reconfigurable systems therefore have the potential to achieve far greater performance than software as a result of bypassing the fetch-decode-execute cycle of traditional microprocessors as well as possibly exploiting a greater degree of parallelism.

A typical reconfigurable architecture contains a software programmable host processor and one or more reconfigurable processing units. The host processor mainly deals with the control tasks and the reconfigurable processing units can be divided into several categories. One is reconfigurable hardware and it mainly processes the data

T. Srikanthan et al. (Eds.): ACSAC 2005, LNCS 3740, pp. 171 – 185, 2005.

operations, another is the reconfigurable bus or interconnection network and the other is the reconfigurable memory and reconfigurable I/O. Among them, the reconfigurable hardware is the most important units of the reconfigurable computing system. It can be classified with three opinions as follows: the granularity of reconfigurable hardware, the static or the dynamic reconfiguration and the reconfigurability. Conventionally, the most common devices used for reconfigurable computing are field programmable gate arrays (FPGAs) [1]. FPGAs allow designers to manipulate gate-level devices such as flip-flops, memory and other logic gates. This makes FPGAs quite useful for complex bit-oriented computations. Examples of reconfigurable systems using FPGAs are [2], [3], [4], and [5]. However, FPGAs have some disadvantages, too. They are slower than ASICs, and have inefficient performance for coarse-grained (8 bits or more) data-path operations. Hence, many researchers have proposed other models of reconfigurable systems targeting different applications. *PADDI* [6], *MATRIX* [7], *RaPiD* [8], *MorphoSys* [12] and *REMARC* [9] are some of 5 the coarse-grain prototype reconfigurable computing systems. Research prototypes with fine-grain granularity (but not based on FPGAs) include *DPGA* [10] and *Garp* [11]. Additionally, there are two mix-grain prototypes include *Pleiades* [13] and *RAW* [14].

In this paper, we design a Fine-grain Multi-context Reconfigurable Processing Unit, called FMRPU. The main advantage of the reconfigurable computing system originates from the reconfigurable processing unit. The reconfigurable computing system can be widely applied in various applications by the characteristics of reconfiguration and achieve the required system performance by exploiting the potential parallelism degree of reconfigurable processing unit. Even if operating at low clock rate, the reconfigurable processing unit can achieve the requirement of real-time processing. Thus the reconfigurable processing unit in the reconfigurable computing system is used to carry out the custom application circuits and to achieve the required processing performance. The design idea of FMRPU is to make use of the reconfigurable operations to provide an eclectic solution between software and hardware.

2 Basic Components of FMRPU

The components of FMRPU include the Logic Cell (LC), Logic Bank (LB), Logic Array (LA), Switch Box (SB) and the Data Stream Switch (DSS). A Logic Bank (LB) comprises of 8 Logic Cells (LCs) and a Logic Array (LA) comprises of 8 Logic Banks (LBs). As illustrated in Fig. 1, the architecture of FMRPU is composed of 8 x 8 Logic Arrays, 16 Switch Boxes (SB) and three Data Stream Switches (DSS). A Fine-grain Multi-context Reconfigurable Processing Cell (FMRPC) contains four LA and a SB which has direct connectivity with these LAs. These components are connected and are able to communicate with each other through three hierarchical-level connectivity.

2.1 Logic Cell (LC)

The LC is the essential unit of reconfiguration, and is also the basic component of FMRPU. Since many applications have a heterogeneous nature and comprise of several

sub-tasks with different characteristics, we require high flexible and reusable hardware resources to map these applications. As illustrated in Fig. 2, a LC consists of the following units: two look up tables (LUTs), one carry chain and cascade chain cell, several multiplexers and a programmable D-flip flop.

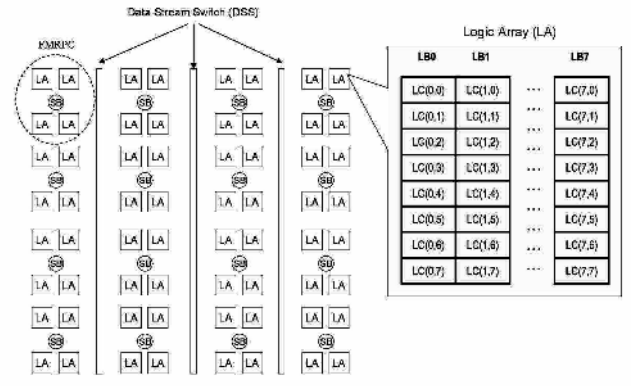


Fig. 1. Block diagram of FMRPU

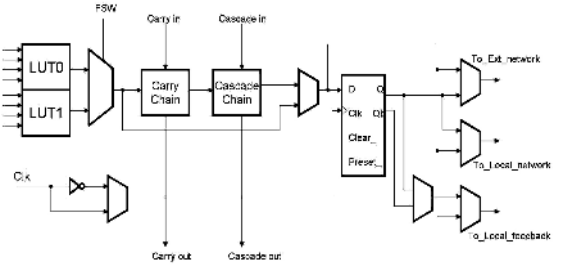


Fig. 2. Block diagram of Logic Cell

A LC contains two 4-input look up tables, and the output of two LUTs is selected by a switch signal "FSW" to perform multi-context switch. Each LUT is capable of implementing any combinational function with up to 4 inputs. The outputs of each unit and the data flow within the LC can be controlled by a set of multiplexers. The output of two LUTs is selected by a switch signal "FSW" to perform multi-context switch to accomplish reconfiguration. When treating LCs as routing resources, the routing path, which bypasses carry chin and cascade chain cell to shorten the latency through a LC, takes 1.8 ns estimated by Synopsys design analyzer for 0.35 micron to pass through a LC.

2.2 Logic Bank (LB) and Logic Array (LA)

A LB is composed of eight LCs as illustrated in Fig. 3. Each LC has a feedback path, and the feedback path allows the local connections to be made from within the LC.

A LA is composed of eight LBs as illustrated in Fig. 4. The LAs of FMRPU are the basic computation cells to be able to perform the various operations by configuration. In FMRPU, there are 64 LAs in all and each can efficiently communicate with others through three hierarchical-level connectivity.

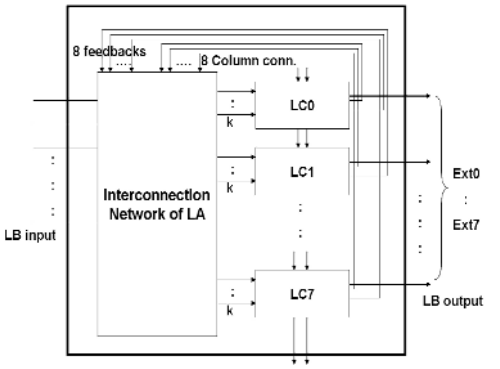


Fig. 3. Logic Bank

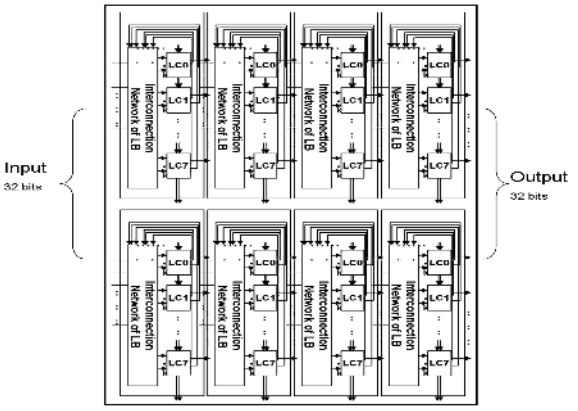


Fig. 4. Logic Array

2.3 Switch Box (SB)

The Switch Box (SB) is located in the center of four LAs, and permits vertical, horizontal, or diagonal connections with these LAs. These four LAs can mutually transmit and receive the material through the SB, and furthermore the SB permits the connections to other neighbor SBs. As illustrated in Figure 5, a SB consists of Input、Output Connection Matrix and four 16 x 16 bits multipliers.

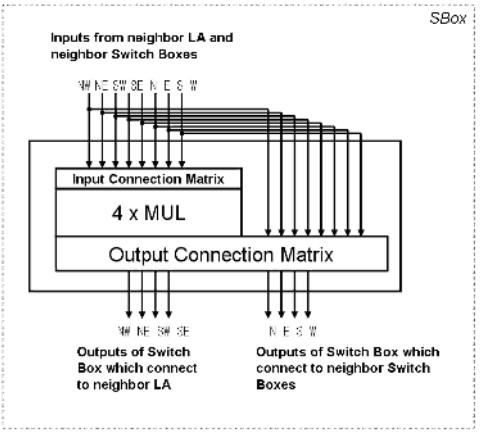


Fig. 5. Switch Box

Each SB has eight channels and the input of each channel is from eight directions. The inputs "NW", "NE", "SW" and "SE" of the SB are from the neighbor LAs and the inputs "N", "E", "S" and "W" of it are from the neighbor SBs. The function of the Output Connection Matrix is the same with the Input Connection Matrix, which is able to rearrange the data streams. The physical delay of Input and Output Connection Matrix is 1.5ns and the delay of multiplier is 6ns estimated by Synopsys design analyzer for 0.35 micron.

2.4 Data Stream Switch (DSS)

Since the target applications may have irregular communication patterns, the routing paths of mapped applications may be irregular and the excessively long routing paths

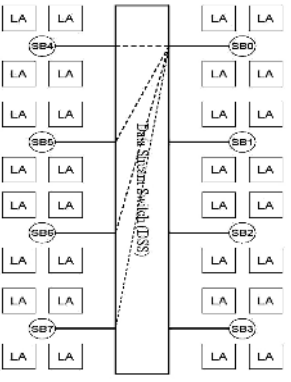


Fig. 6. Data Stream Switch

may limit the performance of mapped applications. The Data stream Switch (DSS) is designed to avoid the situation by rearranging the data streams and to make the mapping of applications more regular to keep off the critical path caused by the global routing paths. As illustrated in Fig. 6, the SBs at the right side of DSS and those at the left side of DSS can access each other through the DSS. For example, the output of SB0 can be accessed by SB4, SB5, SB6 and SB7. Similarly, the output of SB4, SB5, SB6 and SB7 also can be accessed by SB0.

3 Interconnection Network of FMRPU

Since the routing among the LAs and the routing within each LA are two major factors determining the performance of FMRPU, the consideration of interconnection delay is essential to apply performance oriented optimizations to the design of FMRPU. Additionally, we define the **Interconnection Function** and **Neighbor Set** to describe the interconnection network of FMRPU.

3.1 Interconnection Network of LA

The interconnection network of the LA is used to connect 64 LCs, and it determines all the possible connections of a LC. To consider the flexibility and the efficiency, we design an eclectic interconnection network of LA. As shown in Fig. 7, each LC has its coordinate. The intersected LC (0, 1) of two dotted-circles is able to communicate with those LCs in the identical column and row. There are total fourteen LCs to connect to it directly through the interconnection network of LA. The topology of the eight LCs in the same row is the complete connection called as row complete connection, and the topology of those LCs in the same column is also the same called as column complete connection. As shown in Fig. 8, the intersected LC node "(0, 1)" of the row and column complete connection is able to access the output data from 15 LCs in the column "0" and row "1" of LA. The interconnection network of LA is composed of 16 complete connections, include of 8 column complete connections and 8 row complete connections. This interconnection network is named as partial complete connection. The partial complete connection preserves the high flexibility of full connections and the worst routing latency is excellent to some traditional 2-D topologies.

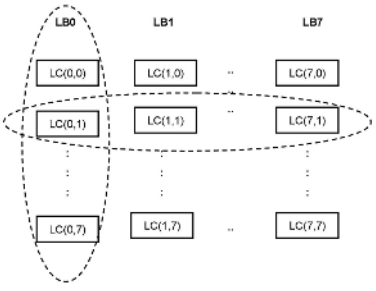


Fig. 7. Interconnection Network of Logic Array

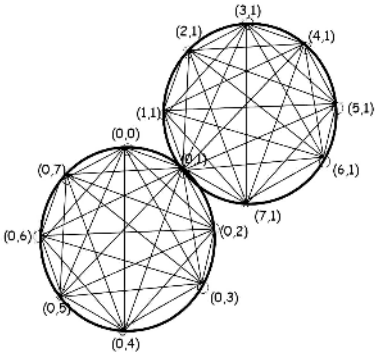


Fig. 8. Topology of Logic Array

3.2 Hierarchical-Level Interconnection Network of FMRPU

The interconnection network of FMRPU is composed of three hierarchical-level connectivity, and each LA of FMRPU is able to communicate with others through the three kinds of connectivity as described below.

(1) Connectivity between Switch Boxes (SB-connectivity): The connectivity throughout all SBs of FMRPU provides nearest neighbor connectivity, which is named as SB-connectivity.

(2) Connectivity among LAs (Row-connectivity): The LA can directly connect to other three LAs in the identical row .The LA at even (odd) position can access the output of other three LAs at the even (odd) position. This kind of connectivity is named as Row-connectivity.

(3) Connectivity between SB and LA (X-net connectivity): The X-net connectivity provides a modified mesh-connected structure suitable for communication among neighboring LAs. A SB, located in the center of four LAs, permits vertical, horizontal, or diagonal connections.

3.3 Hierarchical-Level Interconnection Network of FMRPU

The interconnection network of FMRPU is composed of three hierarchical-level connectivity, and each LA of FMRPU is able to communicate with others through the three kinds of connectivity as described below.

(1) Connectivity between Switch Boxes (SB-connectivity): The connectivity throughout all SBs of FMRPU provides nearest neighbor connectivity, which is named as SB-connectivity.

(2) Connectivity among LAs (Row-connectivity): The LA can directly connect to other three LAs in the identical row .The LA at even (odd) position can access the output of other three LAs at the even (odd) position. This kind of connectivity is named as Row-connectivity.

(3) Connectivity between SB and LA (X-net connectivity): The X-net connectivity provides a modified mesh-connected structure suitable for communication among neighboring LAs. A SB, located in the center of four LAs, permits vertical, horizontal, or diagonal connections.

3.4 Reconfiguration Mechanism of FMRPU

Since FMRPU is designed to be a reconfigurable computing device, the reconfiguration mechanism makes it capable of performing various applications. A distinguishing feature of reconfigurable computing is its ability to configure the computational fabric while an application is running. There are three different basic models of reconfigurable computing**:** single context, multi-context and partial reconfiguration. The reconfiguration mechanism of FMRPU combines two models, which one is multi-context and another is partial reconfiguration. There are some advantages of performing partial reconfiguration on multi-context devices. According to the ability of partial reconfiguration, one advantage is speeding up the reconfiguration rate of background loading. Since full configuration is usually unnecessary when requiring reconfiguration, performing the partial reconfiguration rather than full configuration can accelerate the reconfiguration rate. Furthermore, the mapped hardware circuit can be optimized during the run-time because of the utilization of partial reconfiguration. Thus, performing partial reconfiguration on multi-context devices will add another dimension to the scalability of FMRPU.

According to many applications that require more contexts than the available number of contexts in the device store context configurations in external memory and load them on the device when required, the context switch time in such a system would have a significant latency if the context is not available on the device. Since FMRPU is designed to have two-context device, a context can be configured in the background while another context is processing data. This would reduce the effective context configuration time and hence the average reconfiguration time. Applications which efficiently map to FMRPU may partition into several contexts. To run such applications seamlessly, context switching must occur quickly. The FMRPU has the capability of switching context in a single clock cycle. With these reconfiguration techniques for FMRPU, the reduced reconfiguration time will help the total application execution time to approach the time required for processing data alone.

4 Verification and Analysis

The verification environment includes two parts:

(1)Function verification with FPGA

Verify the functions of each unit and the reconfiguration mechanism of FMRPU by using Altera's QuartusII software.

(2)Logic synthesis with Synopsys design analyzer

Synthesize each component of FMRPU by using Synopsys design analyzer to estimate the hardware complexity and the delay of our design.

Synthesis results

Table 1 shows the delay and area of each unit estimated by Synopsys design analyzer. Table 2 shows the context memory bits required for each component of FMRPU. In addition, we estimate the total area of FMRPU and discuss the full configuration time for a context plane.

Table 1. Area and delay of each unit

Units	Delay(ns)	Gate Count
LC	1.8	110
LB	14.4	1375
LA	115.2	11388
SB	1.5 (for routing) 6.5 (for computing)	11129
DSS	0.5	785

The total SRAM bits of two context planes are 56k bytes, and its area is estimated to be 11.2 mm². The total area of FMRPU is estimated to be 27. 2 mm². The applications mapped to the FMRPU can perform at 110 MHz by taking advantage of the three levels interconnection network and appropriate mapping arrangement. The full configuration time of one plane is 4224 clock cycles, which is significantly long. However, the reconfiguration mechanism of FMRPU can greatly reduce the reconfiguration time and can almost neglect the configuration time by the background loading.

Table 2. Context Memory bits for each unit

Units	Memory (bits)
LA	6932
SB	560
DSS	24

4.1 Mapping to FMRPU

Subsequently, we will show the application mapping to FMRPU and measure the performance of each mapped application. In addition, we will compare the performance of FMRPU with other devices.

(1) Motion Estimation mapped to MPEG

To implement motion estimation in coding image applications, the most popular and widely used method, due to its easy implementation, is the block-matching algorithm (BMA) [18]. The BMA divides the image in squared blocks and compares each block in the current frame (reference block) with those within a reduced area of the previous frame (search area) looking for the most similar one. Among the different block matching methods, Full Search Block Matching (FSBM) involves the maximum computations. However, FSBM gives an optimal solution with low control overhead. Some standards also recommend this algorithm. Full Search Block Matching (FSBM) is formulated using the Sum of Absolute Difference (SAD) criterion in equation (4.1).

$$SAD(m,n) = \sum_{i=1}^{N} \sum_{j=1}^{N} |R(i,j) - S(i+m, j+n)| \qquad (4.1)$$

$$Given\ p \le m, n \le q\ where\ p\ and\ q\ are\ the\ maximum\ displacements.$$

The Equation (4.1) is mapped to FMRPU. Initially, one reference block and the search area associated with it are loaded into one set of the Data Buffer. The FMRPU starts the

matching process for the reference block resident in the Data Buffer. During this computation, another reference block and the search area associated with it are loaded into the other set of Data Buffer. In this manner, data loading and computation time are overlapped.

Using N = 16, for a reference block size of 16 x 16, it takes 40 clock cycles to finish the matching of four candidate blocks. For FMRPU, there are 289 candidate blocks (85 iterations) in each search area, total of 3400 cycles are required to match the search area (FMRPU operates at 110 MHz). For the MorphoSys, it needs 4080 cycles to match the search area. If the reference block size is 8 x 8, the FMRPU takes 13 clock cycles to finish the matching of four candidate blocks. For FMRPU, it needs 85 iterations to match the search area. Therefore, the FMRPU needs 1105 cycles to finish it, and the MorphoSys needs 1133 cycles to finish it. As shown in Fig. 9, the performance of FMRPU is better than MorphoSys. This is due to that FMRPU can match four consecutive candidate blocks concurrently. However, the MorphoSys can only match three consecutive candidate blocks concurrently.

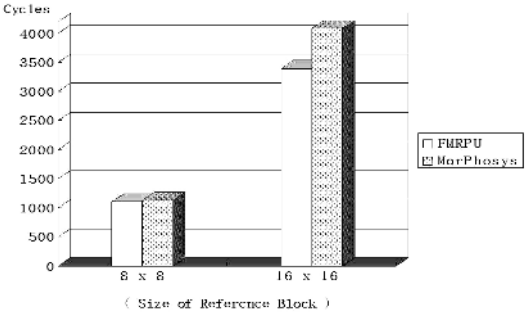


Fig. 9. Performance Comparison of ME

(2) Discrete Cosine Transform mapped to FMRPU

The standard block size for DCT in most image and video compression standards is 8x8 pixels. Since the one DCT Butterfly computation can be directly mapped into a FMRPC (through the X-net connectivity), the FMRPU has the same size as 1-dimension DCT. Initially, an 8x8 block is loaded from the Data Buffer to FMRPU. The data bus between Data Buffer and FMRPU allows concurrent loading of eight pixels. The result of each Butterfly computation is delivered to the next stage via the DSS. Rearrange the data streams to avoid routing the redundant path by the DSS. After four-stage computations, the FMRPU finishes computing the first eight pixels and then sends to the Data Buffer. With the DCT mapping to the FMRPU, eight row (column) DCT computations are computed in parallel.

The 1-D algorithm is first applied to the rows (columns) of an input 8x8 image block, and then to the columns (rows). For FMRPU, it needs 8 clock cycles to complete the 1-D DCT. Assume that for a great deal of blocks, the Data Engine (DEG) firstly fetches the row (column) pixels of these blocks and sends to the Data Buffer to deliver to the FMRPU. After all 1-D row (column) DCT computations are completed, the DEG continues to fetch the column (row) pixels of these blocks to deliver to the FMRPU to

perform the column (row) DCT computations. By overlapping the data storing and computation time we can eliminate the transpose time for 2-D DCT computations. Therefore, the FMRPU requires 16 clock cycles to complete a 2-D DCT computation. The FMRPU needs 16 clock cycles to complete it. This is in contrast to 103 cycles required by Pentium MMX. The MorphoSys needs 21 clock cycles to complete it. The REMARC, which is a coarse-grain reconfigurable computing system, takes 54 cycles for IDCT. A DSP video processor, TI C6200 needs 160 cycles. The relative performance Fig.s for FMRPU and other implementations are given in Fig. 10. Since the operations of IDCT are similar as DCT, we can estimate the computing cost of IDCT is the same as DCT when mapping IDCT to the FMRPU.

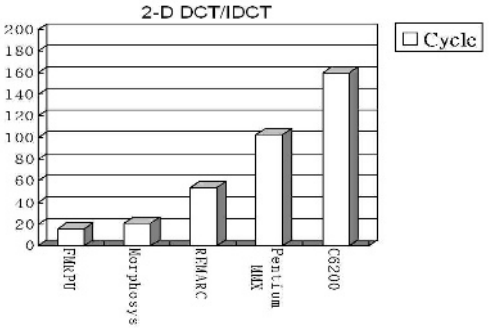


Fig. 10. 2-D DCT/IDCT Performance Comparison

(3) The mapping of FFT, FIR, IIR to FMRPU

FFT: The fast fourier transform (FFT) algorithm is widely used in different areas of application such as communications, radars, imaging etc. The algorithm consists of M= log2 (N) stages and bit-reversing of the output sequence. The Sandy-Tukry butterfly can be mapped to a FMRPC. The four multipliers within a SB can achieve the complex multiplication of FFT. With the number limitation of loaded operands the FMRPU can map three butterfly computations.

For FMRPU, each butterfly operation requires 2 clock cycles. The FMRPU works at 110MHz and it takes 71006 cycles (639.054us) to compute an 8K-point FFT computation, which is satisfied the OFDM in DVB-T spec of 924us. The FMRPU takes 96 cycles to accomplish 2-D FFT with 64 points. This is much less than 276 cycles on C64 and 835 cycles on C62. The performance comparison of the mapping for implementation of 64-point complex 2-D FFT is as shown in Fig. 11.

FIR: The FIR filters are widely used in digital communication systems, in speech and image processing systems and in spectral analysis. It is the sum-of-product processing and each "tap" needs one multiplier and one adder to compute the sampling data and the coefficients. The FMRPU can be configured as a FIR with 16-taps, two FIRs each has 8 taps or four FIRs each have 4 taps. The coefficients are conFig.d into the LAs. Therefore, the FIR can be the low-pass filter, high-pass filter, band-pass filter and band-stop filter by configuring the coefficients in demand. The FMRPU also can simultaneously implement four different kinds of FIR as mentioned above. For FMRPU, each multiply-add operation requires 1 clock cycle, and the mapped FIR can work at 110 Mhz.

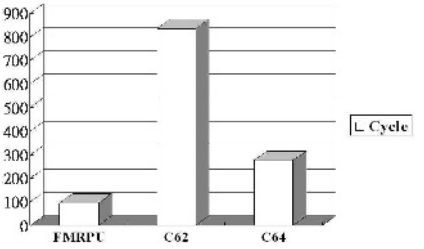


Fig. 11. 2-D FFT Performance Comparison

The performance comparison of the mapping for implementation of 8-taps with 64 output data is as shown in Fig. 12. Because FMRPU has the capability of mapping two 8-taps FIR, the FMRPU only takes 47 cycles to accomplish in this case. This is much less than 269 cycles on C62 and 424 cycles on C67.

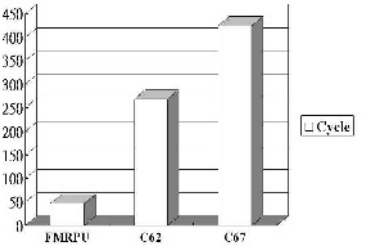


Fig. 12. 8-taps FIR Performance Comparison

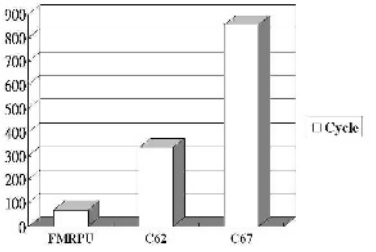


Fig. 13. 4-taps IIR Performance Comparison

IIR: Compare the FIR filters to the IIR filters. The IIR filters require much less memory and fewer arithmetic operations to achieve the same performance. The operations of it are similar as the FIR filters. For each "tap" it requires one multiplier and one adder to compute the sampling data and the coefficients. The sample data are inputted from x(n) and the outputted data are from Y(n). By configuring the coefficients (b0~b6, a0~a6) into the LAs we can implement 6-taps IIR to FMRPU and the IIR filter can operate at 110 MHz.

The performance comparison of the mapping for implementation of 4-taps with 64 output data is as shown in Fig. 13. The FMRPU takes 71 cycles to accomplish in this case. This is much less than 336 cycles on C62 and 855 cycles on C67.

4.2 Performance and Area Analysis of FMRPU

The performance analysis of FMRPU with different size is shown in Fig. 14. The x-axis represents the size of FMRPU and the y-axis represents the speedup which is the performance of FMRPU with different size against to the performance of FMRPU with 2x2 size. There are five algorithms, which include Motion Estimation (ME), DCT, FFT, FIR and IIR, mapped to FMRPU with different size. The input and output data buffer size of FMRPU is set to be 256 bits.

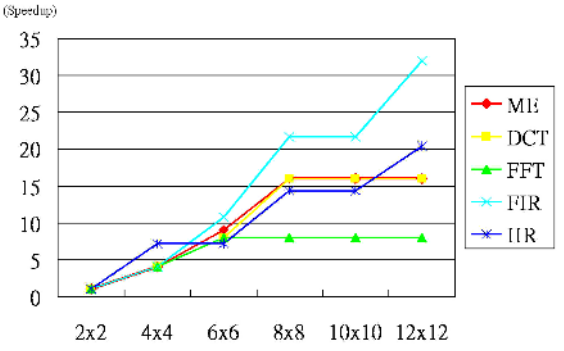


Fig. 14. Performance analysis

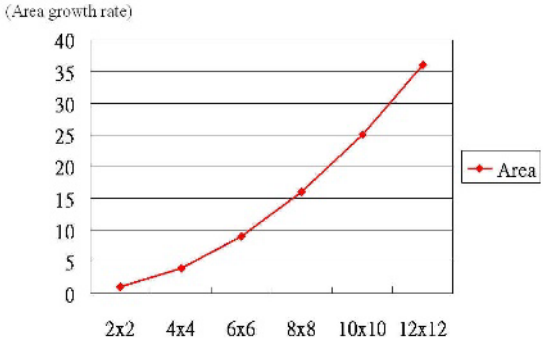


Fig. 15. Area analysis

In Fig. 14, the FFT curve line shows that the performance achieves saturation when the size of FMRPU is 6x6, and this is because of the limitation of input and output data buffer size. The ME and DCT curve lines show that the performance of them achieve saturation when the size of FMRPU is 8x8, and this is because of that the mapping size of two algorithms exactly matches to the 8x8 FMRPU. The FIR and IIR curve lines

show that the performance of them will rise when the size of FMRPU increases. The area analysis of FMRPU with different size is shown in Fig. 15. The x-axis represents the size of FMRPU and the y-axis represents the area growth rate against to the 2x2 FMRPU. We observe the Fig. 14 and Fig. 15 and measure the performance and area size of FMRPU. The appropriate size of FMRPU is 8x8 to perform these algorithms.

5 Conclusion

This paper has presented a novel reconfigurable processing unit, FMRPU. It is designed to have high flexibility due to the reconfigurable components and interconnection network within the FMRPU. It also has high-performance potential for dealing with high-throughput and data-parallel applications. Moreover, the reconfiguration mechanism of FMRPU combines two models: multi-context and partial reconfiguration. It can greatly reduce the effective reconfiguration time. The flexibility and high parallelism degree of FMRPU may make FMRPU widely and more efficiently used in a different application class, such as high-precision signal processing, bit-level computations, control-intensive applications, or dynamic stream processing.

References

[1] S. Brown and J. Rose, “Architecture of FPGAs and CPLDs: A Tutorial,” IEEE Design and Test of Computers, Vol. 13, No. 2 (1996) 42-57
[2] M. Gokhale, W. Holmes, A. Kopser, S. Lucas, R. Minnich, D. Sweely, and D. Lopresti, “Building and Using a Highly Parallel Programmable Logic Array,” IEEE Computer (1991) 81-89
[3] P. Bertin, D. Roncin, and J. Vuillemin, “Introduction to Programmable Active Memories,” in Systolic Array Processors, Prentice Hall (1989) 300-309
[4] S. Hauck, T.W. Fry, M.M. Hosler, and J.P. Kao, “The Chimaera Reconfigurable Functional Unit,” Proceedings of IEEE Symposium on Field-programmable Custom Computing Machines (1997)
[5] M.J. Wirthlin and B.L. Hutchings, “A Dynamic Instruction Set Computer,” Proceedings of IEEE Symposium on Field-programmable Custom Computing Machines (1995)
[6] D. Chen and J. Rabaey, “Reconfigurable Multi-processor IC for Rapid Prototyping of Algorithmic-Specific High-Speed Datapaths,” IEEE Journal of Solid-State Circuits, V. 27, No. 12 (2003)
[7] E. Mirsky and A. DeHon, “MATRIX: A Reconfigurable Computing Architecture with Configurable Instruction Distribution and Deployable Resources,” Proceedings of IEEE Symposium on FPGAs for Custom Computing Machines (1996) 157-66
[8] C. Ebeling, D. Cronquist, and P. Franklin, “Configurable Computing: The Catalyst for High-Performance Architectures,” Proceedings of IEEE International Conference on Application-specific Systems, Architectures and Processors (1997) 364-72
[9] T. Miyamori and K. Olukotun, “A Quantitative Analysis of Reconfigurable Coprocessors for Multimedia Applications,” Proc. of IEEE Sym. on Field-Programmable Custom Computing Machines (1998)

[10] E. Tau, D. Chen, I. Eslick, J. Brown and A. DeHon, “A First Generation DPGA Implementation,” FPD’95, Canadian Workshop of Field-Programmable Devices (1995)
[11] J. R. Hauser and J. Wawrzynek, “Garp: A MIPS Processor with a Reconfigurable Co-processor,” Proc. of the IEEE Symposium on FPGAs for Custom Computing Machines (1997)
[12] Hartej Singh, Ming-Hau Lee, Guangming Lu, Fadi J. Kurdahi, Nader Bagherzadeh “MorphoSys: An Integrated Reconfigurable System for Data-Parallel Computation-Intensive Applications”, University of California, Irvine, CA 92697 and Eliseu M. C. Filho, Federal University of Rio de Janeiro, Brazil
[13] Jan M. Rabaey “Reconfigurable Processing: The Solution to Low-Power Programmable DSP”, Department of EECS, University of California at Berkeley
[14] J. Babb, M. Frank, V. Lee, E. Waingold, R. Barua, M. Taylor, J. Kim, S. Devabhaktuni, and A. Agrawal, “The RAW Benchmark Suite: computation structures for general-purpose computing,” Proc. of IEEE Symposium on Field-Programmable Custom Computing Machines (1997) 134-43
[15] See General Processor Information, http://infopad.eecs.berkeley.edu/CIC/summary/local.
[16] Xilinx, Inc.,” The Virtex Series of FPGAs”. See <http://www.xilinx.com/products/virtex>.
[17] KATHERINE COMPTON Northwestern University AND SCOTT HAUCK University of Washington “Reconfigurable Computing: A Survey of Systems and Software”
[18] César Sanz, Matías J. Garrido, Juan M. Meneses, “VLSI Architecture for Motion Estimation using the Block-Matching Algorithm”, Telecomunicación Technical University of Madrid.
[19] Jim Burns, Adam Donlin, Jonathan Hogg, Satnam Singh, Mark de Wit ,“A Dynamic Reconfiguration Run-Time System”, The Department of Computing Science ,The University of Glasgow Glasgow G12 8RZ, United Kingdom.
[20] Scott Hauck, Matthew M. Hosler, Thomas W. Fry, “High-Performance Carry Chains for FPGAs”, Department of Electrical and Computer Engineering Northwestern University Evanston, IL 60208-3118 USA
[21] KIRAN PUTTEGOWDA , DAVID I. LEHN, JAE H. PARK, PETER ATHANAS, MARK JONES, “Context Switching in a Run-Time Reconfigurable System”, The Journal of Supercomputing, 26 (2003) 239–257
[22] Markus Weinhardt, Wayne Luk, “ Pipeline Vectorization ” ,The 7th Annual IEEE Symposium on Field-Programmable Custom Computing Machines FCCM '99 Marriott at Napa Valley, Napa, California (1999) 20-23
[23] M. I. Masud,"FPGA routing structures: A novel switch block and depopulated interconnect matrix architectures," Master’s thesis, Department of Electrical and Computer Engineering, University of British Columbia (1999)

Modularized Redundant Parallel Virtual File System

Sheng-Kai Hung and Yarsun Hsu

Department of Electrical Engineering,
National Tsing-Hua University, HsinChu 30055, Taiwan
{d897912, yshsu}@ee.nthu.edu.tw

Abstract. Providing data availability in a high performance computing environment is very important, especially in this data-intensive world. Most clusters either use RAID technology or redundant nodes to ensure data availability. People usually use parallel file systems to increase the throughput of a computing system. However, when a parallel file system is involved in a distributed environment, some mechanisms must be provided to overcome the side-effect of using striping. PVFS is a popular and open source parallel file system in the Linux environment, but it provides no fault tolerance. We propose an idea of using distributed RAID technology to ensure the data availability of using striping. By introducing a parity cache table (PCT), we can improve write performance when updating parity is needed. The evaluation of our MRPVFS (Modularized Redundant Parallel Virtual File System) shows that the read performance of MRPVFS is almost the same as that of the original PVFS. As to the write performance, there only exists a little performance degradation due to generating parities.

1 Introduction

In this data-intensive world, it is an increasingly significant requirement to sustain high data throughput in a computing environment. One remedy which comes to mind is using a parallel file system to achieve this. In the past, parallel file systems were proprietary products belonging to specific commercial machines, such as PFS on the Intel Paragon[1], or VESTA[2,3] on the IBM SP2 machine. However, using commodity off the shelf hardware and open source software to construct a high performance computing environment has become popular[4]. Lacking support for parallel file systems, these Beowulf-like clusters cannot gain performance benefits from traditional distributed file systems like NFS[5] even using MPI-IO[6] for data accessing.

PVFS[7] is a practical remedy for providing high performance I/O in the Linux environment. It is good at achieving high read/write throughput and also providing a POSIX interface to legacy applications. However, experience of using PVFS tells us that its reliability is a big problem even if each node's disks use RAID technology[8,9]. This is because PVFS uses a RAID-0 like stripe mechanism across different nodes to enhance I/O performance. Striping is a good

T. Srikanthan et al. (Eds.): ACSAC 2005, LNCS 3740, pp. 186–199, 2005.

technique to allow parallel accesses, but lacking reliability support. Using striping without any fault tolerance technique, the MTTF(Mean Time To Failure) of PVFS may be lower by a factor of $\frac{1}{N}$, where N is the number of I/O nodes in the cluster system. In this situation, what would happen if a disk in one of the nodes fails? All data in the parallel file system cannot be accessed again unless the failed node comes back online and data can be correctly recovered. Things become even worse since not only disk failures must be considered but also other components of a computer may cause a node to fail.

The reliability of PVFS can be improved by purchasing additional disks and using either hardware RAID controllers or software RAID techniques in each of the I/O nodes. This method has two disadvantages, one is the cost incurred, and the other is the MTTF of the overall parallel file system. The first one is obvious since everyone wants to reduce the cost of constructing a COTS (Commodity Off The Shelf) cluster system as much as possible. The second one can be thought of the result of the Amdahl's law[10], since other components combined in a node dominate the node's faults. If one of the nodes in a cluster system using PVFS fails, even if its disks are RAIDed, the striped data within the node can not be accessed unless we replace the broken components with new ones. Using RAID in each node can just only guarantee the availability of data within that node, but what if other components in it fail? This would cause the node to fail and can not participate in the operation of the whole file system. Without the participation of the failed node, data stored in that node can not be accessed and the system can not provide services. In some cases, especially for some server farms, 99.999% availability[11] is required, this means that there only can be a 5 minute down time in a year. To meet this requirement, we must provide a mechanism which can provide high MTTF and recover from failure quickly.

In order to provide high data availability in PVFS, we propose a distributed RAID technique which can increase the MTTF and reduce the impact incurred by parity calculations. The modularized redundant parallel virtual file system (MRPVFS) can not only provide low hardware cost compared with that using RAIDed disks at each node, but also offer higher data availability. RAID-4 technique is used in our system to simplify the overall design. By using a parity cache table, the write performance can be improved and parity information is kept in files within a reasonable data loss rate. This paper is organized as follows: the related research topics are covered in section 2; a quantitative analysis of the overall availability will be presented in section 3; we will show the implementation and evaluation of our MRPVFS versus the original PVFS in section 4; finally we will make some conclusions in section 5.

2 Related Research

In this section, we will present some file systems used in the cluster environment, either distributed or parallel one. Sun Microsystems' NFS (Network File System) is widely used in the traditional UNIX environment, but it lacks support of parallel semantics and the server node is always a single point of failure. It provides

neither fault tolerance mechanism nor striping technique. A well known feature of NFS is its unified interface to user applications. However, its performance is notorious when serving many I/O clients. As we know, some of the clusters in the world still use NFS as their file system and depend on MPI-IO[6] for parallel accesses.

Swift[12] provides conventional file access semantics in its striping file structure. It also supports the same parity method used in RAID level 2 or higher. By the use of parity disks, an error can be recovered as long as it does not happen at the metadata server. Like Swift, Zebra[13] is also such a file system. These two file systems may be called the ancestors of striped file systems in the distributed environment. Berkeley's xFS[14] decentralizes Zebra file system and makes the global file system cache available. xFS was a good research project, but was not further developed due to some license restriction. To avoid the failures of nodes, GPFS[15] and GFS[16] connect disk storages and clients by interconnection switches. This makes the concept of servers disappeared and eliminates the failures caused by servers. However, these proprietary hardware costs more, and can not be applied in the COTS clusters. OSM (Orthogonal Striping and Mirroring)[18] uses the new RAID-x design and provides low software overhead. It enhances high write performance for parallel applications and improves the scalability in the cluster environment.

CEFT-PVFS[17], like PIOUS[19] and Petal[20] provides mirroring (RAID-1 like) to protect data from loss. CEFT-PVFS is based on PVFS and directly implement mirroring over PVFS. It can be regarded as a RAID-10 like parallel file system. Although CEFT-PVFS extends the original PVFS and provides fault-tolerance mechanism, it has some problems. The first is the consistency problem. Since CEFT-PVFS uses mirroring, it needs to guarantee data consistency between working nodes and mirrored nodes. The second one is incurred by the first. Unlike RAID controllers which use bus to transfer data to individual disks, mirroring over the internet relies on the network to transfer data into each node. However, network resource is an important factor that would impact the performance of parallel applications. Using CEFT-PVFS, the network bandwidth would be consumed to some extent. Furthermore, more disk space is needed when using CEFT-PVFS. It wastes 50% of disk space for mirroring.

3 Quantitative Analysis

In this section, we provide MTTF of the three mechanisms used when PVFS is involved. There are: PVFS without any redundant technique ($MTTF_{PVFS}$), PVFS with RAID technique in each node ($MTTF_{PRAID}$), and our modularized redundant PVFS ($MTTF_{MRPVFS}$).

Here, we must define some terms in advance. $MTTF_D$ denotes the mean time to failure of a single disk; N denotes the number of nodes in a cluster; $MTTF_{RAID}$ expresses the MTTF of RAID disks and $MTTF_S$ indicates the MTTF of all other components of a single node in the cluster system.

Since $MTTF_{PVFS}$ just uses striping, its MTTF can be expressed as:

$$MTTF_{PVFS} = 1/(\frac{N}{MTTF_D} + \frac{N}{MTTF_S}) \ . \tag{1}$$

As to $MTTF_{PRAID}$, we can express it as:

$$MTTF_{PRAID} = 1/(\frac{N}{MTTF_{RAID}} + \frac{N}{MTTF_S}) \ . \tag{2}$$

By the analysis of RAID technique in the paper[8], we know that the $MTTF$ of using RAID disks can be expressed as:

$$MTTF_{RAID} = \frac{(MTTF_D)^2}{(D + C *^n G) * (G + C - 1) * MTTR} \ . \tag{3}$$

- D is the total number of data disks
- G means the data disks in a group
- C is the number of check disks *(disks contain parity information)* in a group
- ${}^nG$ denotes the number of groups *(each group has a check disk at least)*
- $MTTR$ means mean time to repair

Using distributed RAID-4 technique in a cluster environment, each I/O node can be used as a parity node or a data node. This tells us that $D + C *^n G = (G+C)*^n G = N$. Using this equation, we can rewrite Equation 3 as Equation 4.

$$MTTF_{RAID} = \frac{(MTTF_D)^2}{N * (\frac{N}{{}^nG} - 1) * MTTR} \ . \tag{4}$$

Applying Equation 4 into Equation 2, we can express the final form of $MTTF_{PRAID}$ as:

$$\begin{aligned} MTTF_{PRAID} &= 1/(\frac{N^2 \times (\frac{N}{{}^nG} - 1) \times MTTR}{(MTTF_D)^2} + \frac{N}{MTTF_S}) \\ &= 1/(\frac{N^2 \times (\frac{N}{{}^nG} - 1) \times MTTR}{(MTTF_D)^2} + \frac{1}{MTTF_{PVFS}} - \frac{N}{MTTF_D}) \\ &\leq 1/(\frac{1}{MTTF_{PVFS}} - \frac{N}{MTTF_D}) \\ &= \frac{MTTF_{PVFS} * MTTF_D}{MTTF_D - N * MTTF_{PVFS}} \\ &= \frac{MTTF_D}{MTTF_D - N * MTTF_{PVFS}} \times MTTF_{PVFS} \ . \end{aligned} \tag{5}$$

Applying Equation 4 in our MRPVFS, we can get $MTTF_{MRPVFS}$. The MTTF of our MRPVFS can be shown in the following equation.

$$MTTF_{MRPVFS} = \frac{(MTTF_{Node})^2}{N * (\frac{N}{{}^nG} - 1) * MTTR} = \frac{\left(\frac{1}{\frac{1}{MTTF_D} + \frac{1}{MTTF_S}}\right)^2}{N * (\frac{N}{{}^nG} - 1) * MTTR} \ . \tag{6}$$

Substituting $MTTF_{PVFS}$ into $MTTF_{MRPVFS}$, Equation 6 can be rewritten as:

$$\begin{aligned} MTTF_{MRPVFS} &= \frac{N^2 * (MTTF_{PVFS})^2}{N * (\frac{N}{{}^nG} - 1) * MTTR} \\ &= \frac{N * {}^nG}{(N - {}^nG)} \times \frac{(MTTF_{PVFS})^2}{MTTR} \\ &\geq \frac{N * {}^nG}{N} \times \frac{(MTTF_{PVFS})^2}{MTTR} \\ &= {}^nG \times \frac{(MTTF_{PVFS})^2}{MTTR} \ . \end{aligned} \tag{7}$$

Table 1. MTTF of these three systems

	MTTF(hours)	Group Size
PVFS (Original)	568	-
PRAID (PVFS with each node equipped with RAID technology)	624	1
MRPVFS (modularize redundant parallel virtual file system)	86088	1

In the real case, MTTF or MTTR is usually measured by hours. If we could make MTTR as small as possible(such as 1 hour), we could rewrite the above equation as.

$$MTTF_{MRPVFS} \geq^{n} G * MTTF^{2}_{PVFS} \ . \tag{8}$$

This means that our modularized redundant parallel virtual file system (MRPVFS) can provide a much better MTTF than the original PVFS or the one with each node's disks RAIDed. Besides, we can also increase ${}^{n}G$ to improve the $MTTF$ of our $MRPVFS$ more. In other words, as the number of nodes increase, the term $MTTF_{PVFS}$ in Equation 8 becomes lower ($MTTF_{PVFS}$ is inversely proportional to N). To maintain an almost constant MTTF when the number of nodes increase, we can just increase the number of parity groups in the system.

For example, MTTF of a disk is no less than 100,000 hours and MTTR is shorter than 4 hours. As to the MTTR of a system, its value is less than that of a disk, since it should account for the failure of power, CPU , network ...etc. We assume that MTTF of a system is usually 10,000 hours. Using the quantity, we can get the MTTF of these three systems in Table 1. Fig. 1 plots

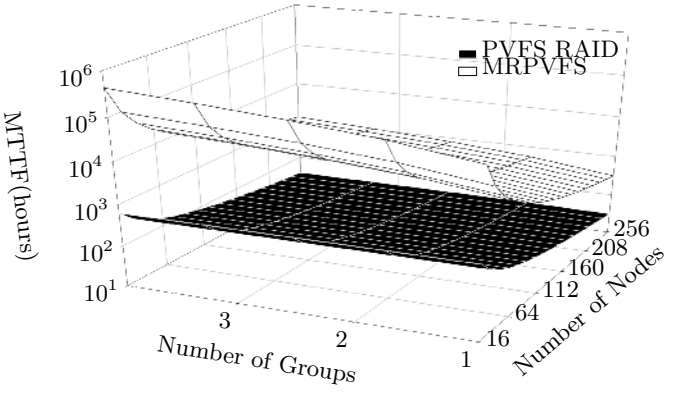


Fig. 1. MTTF of MRPVFS and PVFS RAID: The MTTF of our MRPVFS is better than that of PVFS with each node equipped with RAID technology

the $MTTF$ when more nodes and groups are involved either in $MTTF_{PRAID}$ or $MTTF_{MRPVFS}$.

4 Implementation and Evaluation

As an extension module of PVFS, we implemented the fault tolerance mechanism within PVFS. This implementation includes three parts: parity striping, fault detection, and on-line recovery. We will discuss them in this section. The original striping mechanism of PVFS does not support parity, just like RAID-0. RAID-5 technology has been proven as the best trade-off between performance and efficiency, and should be taken into our consideration when implementing MRPVFS. However, to simplify our design we use RAID-4 instead. Another reason to use RAID-4 as our striping is that updating parity information on our system is not a real-time process, because the information needed is cached and the calculation is thus delayed. This means that we do not encounter the problem of concurrent writes incurred in the RAID-4 structure. By using a parity cache table, we can solve this concurrent write problem in our system. We will describe this algorithm in more detail in the following paragraph.

Fig. 2 shows the normal operation of our MRPVFS. In our MRPVFS, the metadata server is used as a spare I/O node. The spare I/O node stores the parity information calculated whenever a file is created in the system or updating a parity block is required. The SIOD is a spare I/O daemon which comes from the original IOD daemon. SIOD is just used to gather data being read or written from other I/O nodes to clients. The data gathered by SIOD is in turn to produce the parity information. Using RAID technology with parity support, each write process must recalculate the parity. The recalculation process include reading part of the old data needed, XORing this part with newly written data and

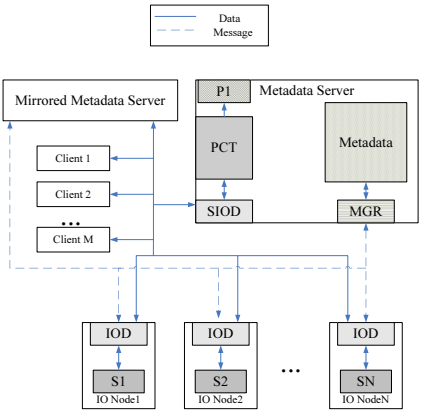


Fig. 2. Normal Operation of MRPVFS: $S1$,$S2$ ... SN represent the stripes of a file, while $P1$ is the corresponding parity. In other words, $S1 \bigoplus S2 \bigoplus \ldots \bigoplus SN = P1$.

finally writing the new data along with parity. Updating parity is a high-cost process, and we propose a technique to alleviate its impact.

4.1 Parity Striping and Updating

Fig. 3 shows the structure of our parity cache table (PCT). In the distributed paradigm, low level block information is meaningless to us. Every disk at each node has its own block sequence used to store a stripe of files. We can not get useful relationship between blocks at different nodes when constructing corresponding parity blocks, unless we perform an initialization process just like what does in a traditional RAID controller. We just use the file offset in different stripes of a file to determine if they belong to the same parity block (i.e. they are used to construct the same parity block). This high level abstraction of blocks can ignore the real block size used in the original file system, and provide heterogeneous environments for different file systems underneath.

As we mentioned before, updating parity is a high-cost process and may reduce the write performance. We construct PCT, like the buffer cache used in traditional Unix file system. The PCT has 4K entries. Each entry contains N (the number of I/O nodes) blocks, a 96-bit tag, and a $2 * \log_2 N$-bit reference count. These N blocks store data blocks needed to construct a parity block. Applications usually read or write more than 1K size, thus a single read or write operation contains more than one block. Whenever a read or write operation happens, the file inode and the read or write offset are used to hash the blocks in the PCT. If there are data already in the PCT blocks, replacements may or may not be required depending on the corresponding reference count. However, if the PCT blocks contain no valid data (i.e. the corresponding reference count field is zero), they are filled with the read or written blocks. A 96-bit tag stores the inode number (64 bits) along with the reduced file offset (32 bits out of 54 bits). It is used to differentiate parity blocks, since different blocks in different files may hash to the same entry. We should compare this tag whenever a read or written block needs to be brought into the PCT blocks. Using this scheme,

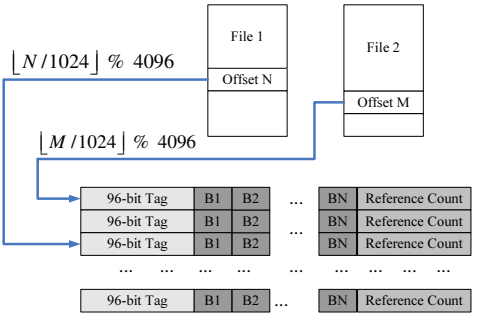


Fig. 3. Parity Cache Table Structure: The offset is used to determine which entry in the PCT should store the corresponding block

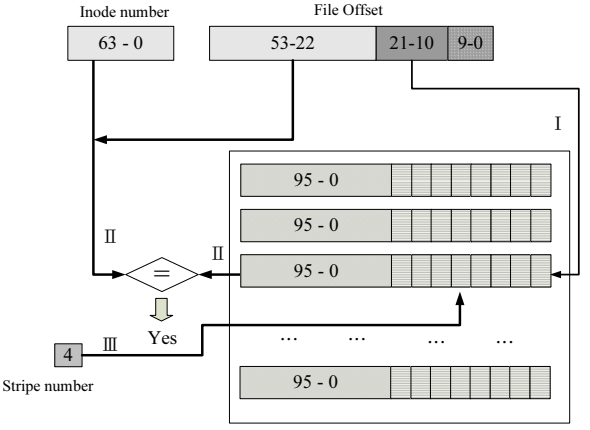


Fig. 4. Hashing and Tag Comparison: The 64-bit inode number and the most significant 32-bit file offset combined to form the 96-bit tag. Tag comparison is needed, since the PCT has a limited entry. The striping number can be determined by the metadata server, and can be directly used to locate the corresponding block. I: Use the fields (10-21 bits) of the file offset to find an entry in the PCT. II: Compare the combined tag (64-bit inode number and most significant 32 bit of the file offset) with the tag field in the PCT entry to see if they are matched. III: The stripe number gotten from the metadata server is used to fetch the block needed.

the maximal size of a file which our MRPVFS can store is $2^{(32+12+10)} = 16$ Peta bytes. Fig. 4 shows the detail process of how to match a block with offset being read or written. Finally, the $2 * \log_2 N$-bit field stores the reference counts in the corresponding parity block. We use high $\log_2 N$ bits to record the number of writes occurred in the corresponding data blocks and the low $\log_2 N$ bits to indicate the number of reads happened. Whenever a read or a write operation happens, the corresponding count is increased by one.

By using PCT, we can aggregate several write operations into one operation. Besides, if some blocks have been in the PCT cached by read operations, we do not need to reread these blocks when calculating parity. This reduces both the number of read and write operations needed when updating a parity.

To prevent data from loss in the PCT, we flush PCT entries back to files under three situations. The first situation occurs when a PCT entry is "replaced". The replacement occurs when the result of hashing points to the nonempty entry in the PCT. Here, the nonempty entry means that its corresponding reference count is larger than zero. The 4K-entries PCT is a limited space, replacement is determined by the $2 * \log_2 N$-bit reference count field. The bigger one is the winner. The loser would write back its parity to disk if its reference count is no less than N (means at least one write operation has occurred). The reference counts of a PCT entry is reset to zero after the parity in it is written back to disk. The second situation occurs when all N blocks of an entry are "ready" for generating a parity block. When all blocks in one of PCT entries contain the

data needed (either a newly written or a previously read block) to perform a parity calculation, the corresponding parity block can be calculated without any extra read operations. In this situation, if the reference count is no less than N, we write back the parity block to disk. The third situation is a periodic "flush". Like bdflush in a Unix file system, we implement a process which periodically flushes PCT entries with reference count no less than N to disk. The period of flush is set to 30 seconds with an average data loss of 15 seconds.

Whenever a flush operation occurs, the being written parity block is also sent to the mirrored metadata server. The mirrored metadata server keeps not only the parity information but also the whole backup of the metadata. The backup process is proceeded by the metadata server itself since it knows when the data has been altered.

4.2 Fault Detection and Online Recovery

PVFS is based on TCP/IP client/server model. Therefore, implementing fault detection is not too difficult. We provide two methods for detecting failures. The first one is "periodic pinging". We do this just as what it does in the iod-ping utility provided in the PVFS package. In the metadata server, we periodically ping each I/O node every 30 seconds to see if any of them fails. The second is "passive discovery". Whenever a client needs to access data, it must query the metadata server to see which I/O nodes containing the data. During this time, the metadata server will check each I/O node to see if any of them fails.

Whenever an I/O node fails, the metadata server starts a normal IOD daemon within it and uses it as the substitute for the broken one. All file operations remain and the I/O clients should not see any difference. The reconstruction of the broken data could be performed whenever data blocks are needed by clients, or it can be recovered totally by the metadata server. We call the former passive recovery and the latter active recovery. We have used the passive recovery method in the current study.

If the metadata sever goes down, it can be replaced with the mirrored one. This process can be made automatically. However, we did not implement this in the current study.

4.3 Performance Evaluation

Table 2 indicates the platform hardware used in the evaluation experiments. Each node has a single AMD Athlon XP 2400+ CPU, except for the metadata server. The metadata server has dual AMD Athlon XP 2400+ CPUs inside. The

Table 2. Evaluation Platform

	Disk Size	Memory Size
Metadata Server	160 GB SCSI Disk	1 GB
IO Node	4 GB IDE Disk	512 MB
Client node	4 GB IDE Disk	512 MB

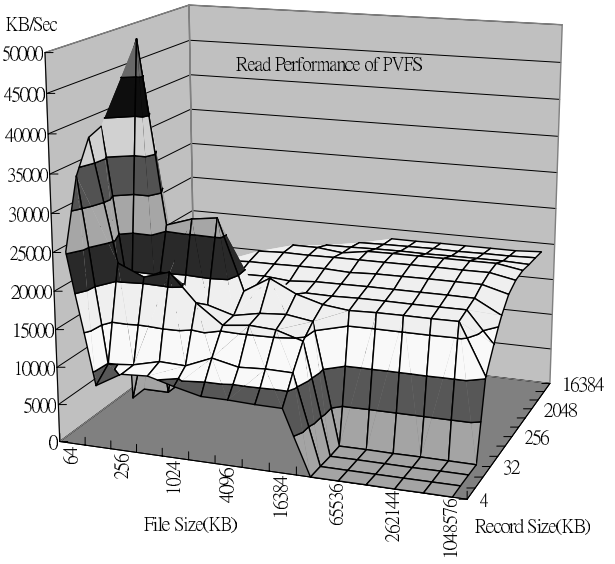


Fig. 5. Read Performance of PVFS : The read performance is measured by one single client and eight I/O servers. Kernel mode PVFS library is used to let IOzone benchmark program perform test without any modification.

operating system is Redhat Linux 7.3 with kernel version 2.4.20 in each node. In our evaluation platform, eight I/O nodes are used with a single I/O client. Each test is based on the traditional TCP/IP network with 100 Mbps Ethernet card. We use the file system benchmark IOzone[21] to perform the test, each test is through the POSIX interface with default striping size of 64 KB. Each test contains file size ranging from 64 KB to 1 GB with different record size ranging from 4 KB to 16 MB.

Fig. 5 shows read performance of the original PVFS, while the read performance of our MRPVFS is shown in Fig. 6. The read operations do not need to recalculate the parity information, and thus incur no performance loss due to writes. The little lower read performance of our MRPVFS is caused by the extra socket connections between the I/O nodes and the SIOD daemon for the transfer of the data blocks. The PCT has no impact on read performance since the read data is only placed in PCT with replacement or not. The read operation is done when the data read is sent to the PCT. This cause a little performance degradation. In addition, no flush operations occur since the reference count of each entry is less than N. The higher peaks on the left side of both figures show the effect of local CPU cache. The lower region at the bottom-right side exhibits the real disk performance. We can just notice the flat region of both figures to see the difference of these two different mechanisms.

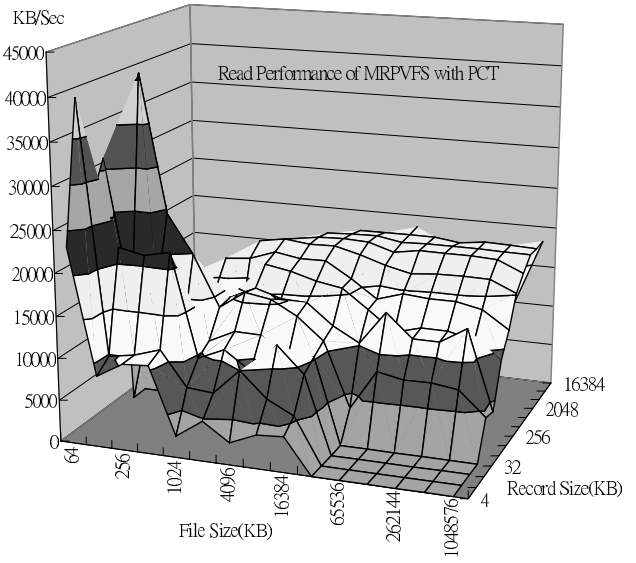


Fig. 6. Read Performance of MRPVFS with PCT support: Compared with Fig. 5, the little lower read performance of our MRPVFS is caused by the extra socket connections between the I/O nodes and the SIOD daemon for the transfer of the data blocks

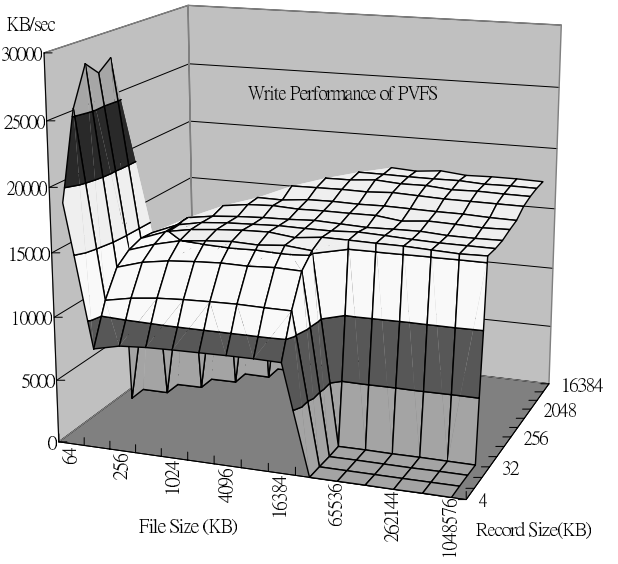


Fig. 7. Write Performance of PVFS: The write performance of PVFS is smooth due to no parity updating required

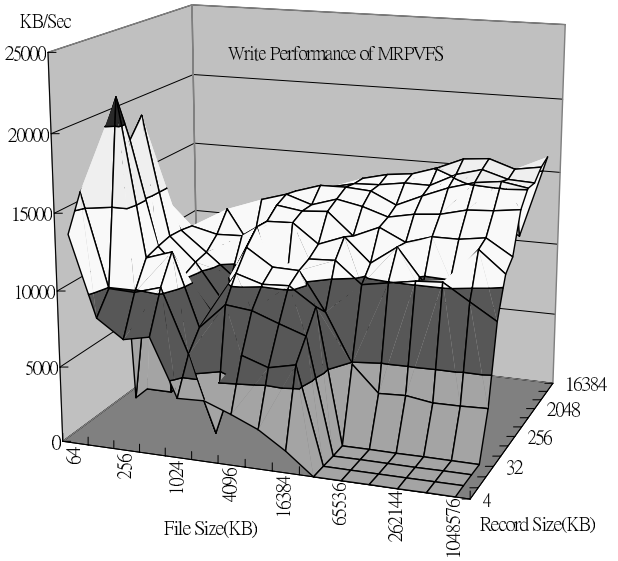


Fig. 8. Write Performance of MRPVFS without PCT: Parity updating occurs frequently since every write requires a modification of parity. This can be shown by the hills of the figure above. In other words, it means that the write performance in RAID-4 is varied due to concurrent writes.

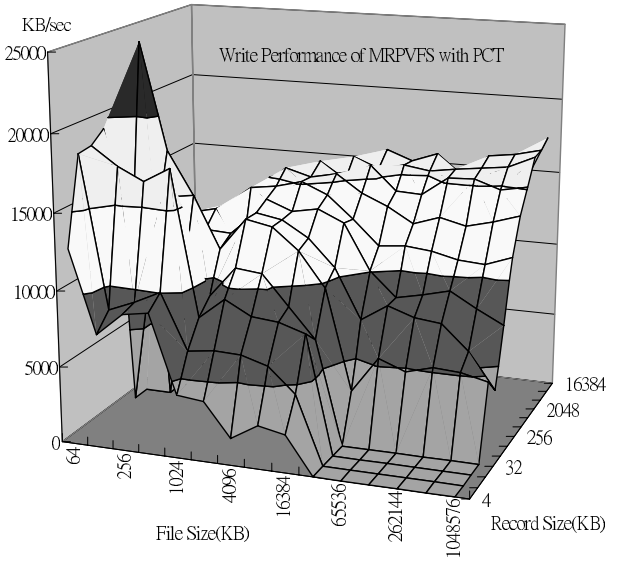


Fig. 9. Write Performance of MRPVFS with PCT: With the support of a parity cache table, the problem of concurrent writes can be alleviated. Compared with Fig. 8, the figure shows a smother result and exhibits better performance.

Fig. 7, 8, 9 show the write performances of PVFS, MRPVFS, MRPVFS with PCT support. PVFS does not need to update parity blocks, its write performance is much more smooth. In our MRPVFS, the write operation is done when the corresponding parity blocks have been written. But whether the real write is done or not depending on the flush mechanism and the buffer cache design of the file system underneath. From Fig. 8, we can see that without PCT support, parity updates occur frequently. This in turn causes the write performance to be varied (the hills in the figure). As to our MRPVFS with PCT support, the parity updating operation has been modified. Parity blocks are actually written only when PCT entry is "ready" or needs to be replaced and its reference count is larger than N. In effect, the Parity Cache Table can dramatically reduce the number of parity writes and greatly improve write performance. Therefore, the throughput of our MRPVFS with PCT support is almost the same as that of PVFS in the flat region, since we reduce the numbers of flush operations needed whenever there is a write operation.

5 Conclusions

We have successfully designed and implemented a modularized redundant parallel virtual file system (MRPVFS) which can continue to supply data to requesting clients even if one of the I/O nodes fails due to any reason. Our MRPVFS can provide almost the same read performance compared with PVFS. As to the write performance, with the help of PCT, the write performance is also comparable.

References

1. Intel Supercomputer System Division, "Intel Paragon Users' Guide," June 1994
2. P.F Corbett, D.G. Feitelson, J.-P. Prost, G.S. almasi, S.J. Baylor, A.S. Bolmaricich, Y. Hsu, J. Satran, M. Sinr, R. Colao, B.Herr, J. Kavaky, T.R. Morgan and A. Zlotel, "Parallel File Systems for IBM SP Computers," *IBM Systems Journal,* Vol. 34, No. 2, pp. 222-248, 1995.
3. P.F. Corbett and D.G. Feitelson, "The Vesta Parallel File System," *ACM Trans. on Computer Systems,* Vol. 14, No. 3, pp. 225-266, Aug. 1996.
4. "The beowulf cluster," available at http://www.beowulf.org/.
5. S. Kleiman D. Walsh R. Sandberg, D. Goldberg and B. Lyon, "Design and Implementation of the Sun Network File System," *Proc. Summer USENIX Technical Conf.,* pp. 119-130, 1985.
6. Message Passing Interface Forum, "MPI2: Extensions to the Message Passing Interface," 1997.
7. Robert B. Ross Philip H. Carns, Walter B. Ligon III and Rajeev Thakur, "PVFS: A Parallel File System for Linux Clusters," *Proc. 4th Annual Linux Showcase and Conference,* USENIX Association, pp. 317-327, Atlanta GA, 2000.
8. Garth Gibson. Randy H. Katz David A. Patterson, "A Case for Redundant Arrays of Inexpensive Disks (RAID)," *Int'l Conf. on Management of Data (SIGMOD),* pp. 109-116, 1988.
9. Jehan-François Pâris, "A Disk Architecture for Large Clusters of Workstations," *Cluster Computing Conf.,* GA, Mar. 1997

10. G.M. Amdahl, "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities," *Proc. AFIPS Conf.,* vol. 30, pp. 483-485, 1967.
11. Brown. A. and D. A. Patterson, "Embracing Failure: A Case for Recovery-Oriented Computing (ROC)," *High Performance Transaction Processing Symp.,* Asilomar, CA, Oct. 2001.
12. D. D. E. Long, B. R. Montague and L.-F. Cabrera, "SWIFT/RAID: A Distributed RAID System," *Computing Systems,* Vol. 7, No. 2, pp. 333-359, 1994.
13. John H.Hartman and John K.Ousterhout, "The Zebra Striped Network File System," *Proc. of the Fourteenth ACM Symp. on Operating Systems Principles,* pp. 29-43, 1993.
14. Curtis Anderson Wei Hu Mike Nishimoto Adam Sweeney, Doug Doucette and Geo Peck, "Scalability in the xFS File System," *USENIX Annual Technical Conference,* 1996.
15. Frank Schmuck and Roger Haskin, "GPFS: A Shared-Disk File System for Large Computing Clusters," *Proc. File and Storage Technologies Conf. (FAST' 02),* pp. 231-244 Jan. 2002.
16. K. Preslan M. O' Keefe S. Soltis, G. Erickson and T. Ruwart, "The Global File System: A System for Shared Disk Storage," *IEEE Trans. on Parallel and Distributed Systems,* Oct. 1997.
17. Yifeng Zhu, Hong Jiang, Xiao Qin, Dan Feng, and David R. Swanson, "Design, Implementation and Performance Evaluation of a Cost-Effective, Fault-Tolerance Parallel Virtual File System," *Int'l Workshop on Storage Network Architecture and Parallel I/O,* Sept. 2003.
18. Kai Hwang, Hai Jin and Roy S.C. Ho, "Orthogonal Striping and Mirroring in Distributed RAID for I/O-Centric Cluster Computing," *IEEE Trans. on Parallel and Distributed Systems,* Vol. 13, No. 1, pp. 26-44, Jan. 2002.
19. S. A. Moyer and V. S. Sunderam, "PIOUS: A Scalable Parallel I/O System for Distributed Computing Environments," *Proc. Scalable High-Performance Computing Conf.,* pp. 71-78, 1994.
20. Edward K. Lee and Chandramohan A. Thekkath, "Petal: Distributed Virtual Disks," *Proc. Seventh Int'l Conf. Architectural Support for Programming Languages and Operating Systems*, pp. 84-92, 1996.
21. "Iozone Filesystem Benchmark," available at http://www.iozone.org/

Resource-Driven Optimizations for Transient-Fault Detecting SuperScalar Microarchitectures

Jie S. Hu[1], G.M. Link[2], Johnsy K. John[1],
Shuai Wang[1], and Sotirios G. Ziavras[1]

[1] Electrical and Computer Engineering, New Jersey Institute of Technology, Newark, NJ 07102, USA
[2] Microsystems Design Lab, The Pennsylvania State University, University Park, PA 16802, USA

Abstract. Increasing microprocessor vulnerability to soft errors induced by neutron and alpha particle strikes prevents aggressive scaling and integration of transistors in future technologies if left unaddressed. Previously proposed instruction-level redundant execution, as a means of detecting errors, suffers from a severe performance loss due to the resource shortage caused by the large number of redundant instructions injected into the superscalar core. In this paper, we propose to apply three architectural enhancements, namely 1) floating-point unit sharing (FUS), 2) prioritizing primary instructions (PRI), and 3) early retiring of redundant instructions (ERT), that enable transient-fault detecting redundant execution in superscalar microarchitectures with a much smaller performance penalty, while maintaining the original full coverage of soft errors. In addition, our enhancements are compatible with many other proposed techniques, allowing for further performance improvement.

1 Introduction

Transient hardware failures due to single-event upsets (SEUs) (a.k.a., soft errors) are becoming an increasing concern in microprocessor design at new technologies. Soft errors happen when the internal states of circuit nodes are changed due to energetic particle strikes such as alpha particles (emitted by decaying radioactive impurities in packaging and interconnect materials) and high-energy neutrons induced by cosmic rays [20]. Technology trends such as continuously reducing logic depths, lowering supply voltages, smaller nodal capacitances, and increasing clock frequency will make future microprocessors more susceptible to soft errors. As such, future processors must not only protect memory elements against soft errors with error checking and correcting codes such as parity and ECC, but also protect the combinational logic within the data path in some fashion [15]. Consequently, previous proposals have suggested utilizing the inherent resource redundancy in simultaneous multithreading (SMT) microprocessors and chip-multiprocessors (CMP) to enhance the datapath reliability with concurrent

T. Srikanthan et al. (Eds.): ACSAC 2005, LNCS 3740, pp. 200–214, 2005.

error detection [12,18,11,19,4,7]. Some recent research has proposed that designers exploit the redundant resources in high-performance superscalar out-of-order cores to enable a reliable processor through instruction-level redundant execution [6,10,9,17]. Most of these techniques execute redundant copies of instructions and compare the results to verify the absence of single-event upsets. The goal of such a reliable datapath design is to provide affordable reliability in microprocessors with minimized cost and complexity increase. However, instruction-level redundant execution causes a severe performance penalty (up to 45% in dual-instruction execution (DIE) for a set of SPEC95 and SPEC2000 benchmarks [10]) as compared to an identically configured superscalar core without any redundant execution. This performance loss is mainly due to resource shortages and contention in functional units, issue bandwidth, and the instruction window, caused by the large number of redundant instructions being injected into the superscalar core [17]. Therefore, to provide the perfect soft-error coverage with minimum performance loss and minimum hardware changes is the major challenge in transient-error detecting dual-instruction execution designs.

In this paper, we first analyze the resource contention on different types of functional units in dual-instruction execution (DIE) environment. Our analysis indicates that different types of functional units receive very different contention and that some applications do not cause significant contention on functional units in redundant execution. The performance of integer benchmarks, especially those with high and moderate IPC (instructions per cycle), is significantly constrained by the available integer ALUs. On the other hand, most low IPC floating-point and integer benchmarks suffer more performance loss from the reduced instruction window size due to the redundant instructions. Based on these observations, we propose applying three architectural enhancements for instruction-level redundant execution, namely 1) floating-point unit sharing (FUS), 2) prioritizing the primary instructions (PRI), and 3) early retiring (ERT) of redundant instructions, for full protection against transient faults with negligible impact on area and a much lower performance overhead.

Our floating-point unit sharing (FUS) exploits the load imbalance between integer ALUs and floating-point ALUs and alleviates the pressure on the integer ALUs in DIE environment by offloading integer ALU operations to idle floating-point ALUs. Different from a previous compiler scheme [13], floating-point unit sharing utilizes a conventional unified instruction scheduler to issue operations to both integer and floating-point functional units. Prioritizing the primary stream (PRI) for instruction scheduling exploits the fact that the duplicate copy can directly use the results computed from the primary execution and form asymmetric data dependences between the duplicate stream and the primary stream. PRI has the ability to restrain the duplicate instructions (along the mispredicted path) from competing with the primary instructions, thus has the potential to improve performance in DIE. As dual-instruction execution effectively halves the size of the instruction window, it may significantly reduces the ability of the processor to extract higher ILP (instruction level parallelism) at runtime. This is especially true for most low IPC floating-point benchmarks, as they rely on a large

instruction window to expose available ILP. The early-retiring (ERT) approach exploits the fact that it is not necessary to keep all redundant copies occupying the instruction window once a redundant instruction is issued for execution. ERT immediately removes the redundant instruction (either the original or the duplicate copy) from the instruction window after it is issued to increase the effective window size, and keeps the last issued redundant copy of that instruction in the window until the commit stage, allowing result comparison before retiring the instruction from the instruction window. Our experimental evaluation using SPEC2000 benchmarks shows these three proposed enhancements, when combined, reduce the performance penalty of dual-instruction execution (DIE) from 37% to 17% for floating-point benchmarks, and from 26% to 6% for integer benchmarks, allowing it to be used in a wide variety of situations without significant performance penalties. In addition, our enhancements are compatible with many other proposed techniques, allowing for further performance improvement.

The rest of the paper is organized as follows. Section 2 reviews the background of dual instruction execution and presents our experimental framework. A detailed analysis of resource competition on different functional units, issue bandwidth, and the instruction window is given in Section 3. In Section 4, we propose three architectural enhancements, floating-point unit sharing (FUS), prioritizing the primary instructions (PRI), and early retiring (ERT) of redundant instructions for performance recovery in DIE. We evaluate our proposed schemes for dual-instruction execution in Section 5 and discuss related work in Section 6. Section 7 concludes this paper.

2 Baseline Microarchitectures and Experimental Framework

2.1 Basics About SIE and DIE

We use the same terminologies as in [9], where SIE (single instruction execution) refers to the baseline superscalar processors without any redundant instruction execution, and DIE (dual instruction execution) refers to the same processor while executing a redundant copy of the original code for transient-error detection. To support two-way redundant execution in DIE, each instruction is duplicated at the register renaming stage to form two independent instruction streams (the primary and duplicate instruction streams) that are scheduled individually. The register renaming logic only needs a very slight change to support DIE, as the physical register number to be assigned to an operand in the duplicate instruction is implied by the corresponding field in the primary instruction [10]. At commit stage, the results of these two redundant instructions are compared before retirement to verify the absence of soft errors. If the two results do not match, a soft error is implied, and a recovery must be initiated. The datapath of a DIE superscalar processor is given in Figure 1. The shaded area sketches the *sphere of replication* in DIE datapath. All other components outside this sphere are assumed to be protected by other means such as information

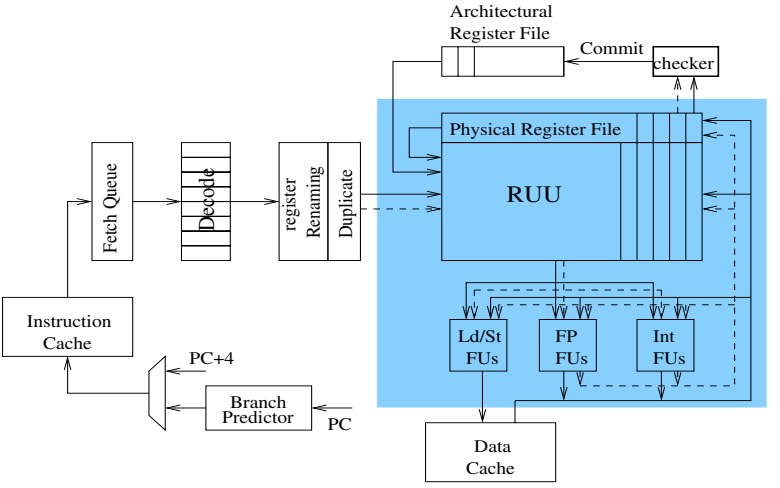


Fig. 1. Datapath of DIE superscalar processors for current transient error detection as proposed in [10]. The Sphere of Replication is sketched by the shaded area.

redundancy (using parity, ECC, etc.). Therefore, a full coverage of soft errors is achieved for the superscalar datapath through the DIE mechanism. However, this full coverage comes at the cost of severe performance loss (from 2% to 45% for SPEC2000 benchmarks as reported in [10]) due to the hardware resource contention and shortage caused by redundant instruction execution.

2.2 Experimental Setup

Our simulators are derived from SimpleScalar V3.0 [3], an execution-driven simulator for out-of-order superscalar processors. In our implementation, the physical register file is separated from the register update unit (RUU) structure. The base-

Table 1. Parameters for the simulated baseline superscalar processor SIE

Parameters	Value
RUU size	128 entries
Load/Store Queue (LSQ) size	64 entries
Physical Register File	128 registers
Fetch/Decode/Issue/Commit Width	8 instructions per cycle
Function Units	4 IALU, 2 IMULT/IDIV, 2 FALU, 1 FMULT/FDIV/FSQRT 2 Mem Read/Write ports
Branch Predictor	combined predictor with 4K meta-table, a bimodal predictor with 4K table, and a 2-level gshare predictor with 12-bit history BTB: 2048-entry 4-way, RAS: 32-entry
L1 ICache	64KB, 2 ways, 32B blocks, 2 cycle latency
L1 DCache	64KB, 2 ways, 32B blocks, 2 cycle latency
L2 UCache	1MB, 4 ways, 64B blocks, 12 cycle latency
Memory	200 cycles first chunk, 12 cycles rest
TLB	4 way, ITLB 64 entry, DTLB 128 entry, 30 cycle miss penalty

line superscalar processor, which model a contemporary high-performance superscalar processor, (SIE) is configured with parameters given in Table 1.

For experimental evaluation, we use a representative subset of SPEC2000 suite, including 10 integer benchmarks and 7 floating-point benchmarks. The SPEC2000 benchmarks were compiled for Alpha instruction set architecture with "peak" tuning. We use the reference input sets for this study. Each benchmark is first fast-forwarded to its early single simulation point specified by SimPoint [14]. We use the last 100 million instructions during the fast-forwarding phase to warm-up the caches if the number of skipped instructions is more than 100 million. Then, we simulate the next 100 million instructions in details.

3 Redundant Instruction Execution Analysis and Motivation

This section presents our detailed analysis of resource contention on different types of functional units and motivates our proposals for performance improvement in redundant instruction execution. A performance comparison between DIE and SIE given in Figure 2 shows that, across all these benchmarks, DIE loses performance up to 44% in floating-point benchmarks and up to 38% in integer benchmarks, compared to SIE. On the average, the performance loss of DIE vs. SIE is 37% for floating-point benchmarks and 26% for integer benchmarks. Note that both SIE and DIE have an issue width of 8 instructions per cycle and 11 functional units including memory Read/Write ports as given in Table 1.

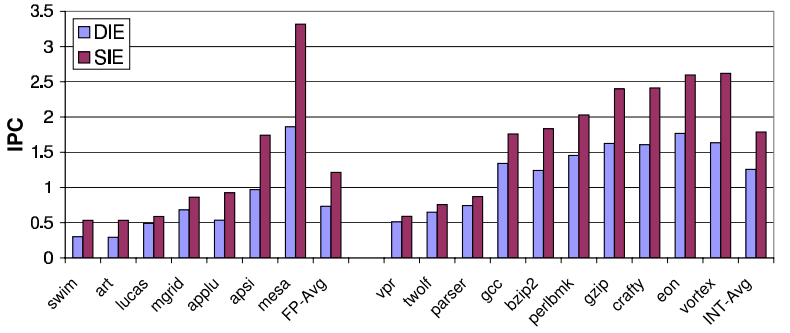


Fig. 2. Performance loss due to dual-instruction execution: DIE vs. SIE

3.1 Resource Contention on Different Functional Units

The functional unit pool in the simulated SIE and DIE processors consists of 4 integer ALUs, 2 integer multiplier/dividers, 2 floating-point ALUs, 1 floating-point multiplier/divider/sqrters, and 2 memory read/write ports. Integer ALUs

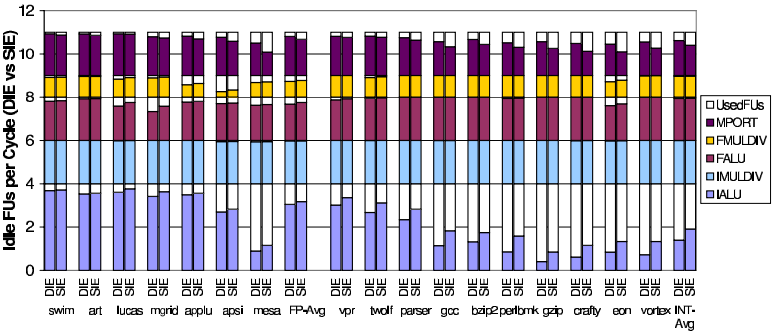


Fig. 3. Functional unit idleness/utilization comparison between DIE and SIE

perform following operations: arithmetic, compare, move, control, memory address calculation, logic, shift, and byte manipulation. All operations are pipelined except division and sqrt operations. Obviously, a scheme trying to double the effective bandwidth of all functional units might be an overkill since not all the functional units are the contenting resources in DIE. Thus, the critical question we are going to answer here is: *what is the resource contention on these five different functional units?*

To better understand the resource contention caused by different instructions, Figure 3 gives the average idleness/utilization per cycle for each of these five different functional units in DIE comparing to SIE. The clear message conveyed through Figure 3 is that the integer ALUs are the most heavily used functional units for both integer benchmarks and floating-point benchmarks with moderate and high IPC (> 1.5). Notice that the integer ALU utilization in these benchmarks are more than 2 in SIE, which means they will require more integer ALUs than the available 4 integer ALUs in DIE in order to avoid performance loss. Consequently, integer ALU contention will be the big performance hurdle in DIE for those benchmarks provided with enough issue width. On the other hand, low IPC benchmarks do not put too much pressure on integer ALUs in both SIE and DIE, which indicates that the integer ALU contention is not the major cause of performance loss in DIE. Another important observation from Figure 3 is that the majority of floating-point ALUs are remaining idle during the simulation of both integer and floating-point benchmarks. If we can utilize these idle floating-point ALUs to take off some load from the heavily used integer ALUs in high IPC applications, the performance of DIE can be improved by providing a much larger bandwidth for integer ALU operations.

3.2 Competition on Issue Bandwidth

Besides functional unit contention, the increased competition on issue ports poses another source of performance loss in DIE. However, issue port competition is closely related to functional unit contention. Doubling the issue width alone without increasing the functional unit bandwidth or instruction window size has

negligible impact on performance [9]. Figure 3 also gives the average raw numbers of instructions issued per cycle (the sum of used functional units) for DIE and SIE. Except for few benchmarks such as *mesa* (4.5 instructions per cycle) and *eon* (4.1 instructions per cycle), the raw number of issued instructions per cycle is lower than 4 in SIE. Notice that the issue width is 8 instructions per cycle in the simulated SIE and DIE processors. This raises the question:*can we effectively use the remaining issue bandwidth for the redundant stream in DIE?* This can be achieved by an enhanced DIE microarchitecture that forms dependences between the primary and duplicate streams and prioritizes instructions in the primary stream at issue stage, which is detailed in Section 4.2.

3.3 Impact of Half-Sized Instruction Window in DIE

The size of the instruction window in DIE is effectively halved when compared to the instruction window in SIE. Superscalar architectures rely on the instruction window to extract ILP of applications at runtime. Applications that need to maintain a large instruction window to expose ILP are very sensitive to the window size variation. In general, a size reduction in the instruction window prevents available ILP from being extracted and thus hurts performance. That is also one of the reasons why the functional units and issue bandwidth can not be saturated in DIE as shown in Figure 3. More importantly, the performance loss of low IPC floating-point and integer benchmarks in DIE is not likely due to the contention on functional unit bandwidth or issue width as indicated by Figure 2 and Figure 3. Instead, we suggest the half-sized instruction window be the performance killer for those low IPC benchmarks in DIE. In this work, we exploit the redundant nature of the duplicate instructions in the instruction window to effectively increase the window size for redundant execution.

4 Resource-Driven Optimizations for DIE

4.1 Exploiting Idle FP Resources for Integer Execution

As modern processors are expected to execute floating-point operations as well as integer operations, most commercial processors have a number of floating-point ALUs available on chip. Our analysis of functional unit utilization in the previous Section indicates that the integer ALUs bandwidth presents a major performance bottleneck for high IPC applications in DIE, while most of the floating-point ALUs are in idle state. We adopt the idea of offloading integer ALU operations to floating-point ALUs in [13] to exploit idle floating-point ALUs thus reducing the pressure on integer ALUs in DIE. In [13] the compiler takes full responsibility for identifying code segment (integer operations) to be executed in floating-point units and inserting special supporting instructions (in need of instruction set extensions) to control the offloaded integer instructions. However, our scheme, called floating-point unit sharing (FUS), is a pure hardware implementation. We exploit a unified instruction scheduler that issues instructions to both integer and

floating-point functional units. In a value-captured scheduler, the results along with tags are broadcasted back to the instruction window at writeback stage. In FUS, only duplicate instructions are allowed to be executed by floating-point units during the lack of integer ALUs. When offloading to floating-point units, the integer ALU instructions are assigned an additional flag to direct the floating-point units to perform the corresponding integer operations.

Since modern microprocessors already have their floating-point units supporting multimedia operations in a SIMD fashion [5], the hardware modifications required to support FUS are minimal. Basically, there are two ways to augment the floating-point ALU for integer ALU operations. One is to slightly modify the mantissa adder to support full-size integer ALU operations and bypass all unnecessary blocks such as mantissa alignment, normalization, and exponent adjustment, during integer operations. The second choice is to embed the simple small integer unit within the much larger floating-point unit. The hardware costs for both these two options at today's technology should be small. Once an integer ALU operation is issued to a floating-point unit, it either has its source operands ready (stored in its RUU entry) while waiting in the issue queue or obtains operands from the bypassing network. In the first case, the opcode and source operand values are read out from RUU entry and sent to the functional unit simultaneously. However, the latter case will require bypassing path from integer units to all floating-point ALUs and vice versa. Remember that FUS only allows the duplicate copy to be issued to the floating-point units, and with prioritizing the primary stream discussed in the next subsection, the duplicate instructions do not have out dependences and do not need to broadcast results to the issue queue. Thus, there are two design choices concerning the changes to the bypassing network. First choice is to augment the existing bypassing network with additional path from integer functional units to floating-point ALUs. The second one is to issue only duplicate instructions with already available source operands to floating-point ALUs or delay their issue until the issue queue captures all the source operands. In our design, we choose the first choice for performance sake.

4.2 Prioritizing Instructions in the Primary Stream

We observed that the two redundant streams in DIE do not have to be independent of each other in order to provide full coverage of soft errors in the sphere of replication. The duplicate stream can directly receive source operand values from the primary stream (results produced) rather than itself since these result values will finally be verified when the producer instruction is committed. Thus the duplicate instructions form dependences on instructions in the primary stream rather than within its duplicate stream. More importantly, the duplicate instruction do not need to maintain out dependences, which means the duplicate instructions produce results only for soft-error checking against the primary copies. Since both the primary and duplicate streams are data dependent on instructions in the primary stream, the primary instructions should be issued as soon as possible for performance consideration. Once the primary instruction

is issued, it wakes up instructions in both redundant streams. Delaying duplicate instructions from issue does not hurt DIE performance since no instruction depends on the results of duplicate instructions, unless it blocks instructions from retiring. Based on this observation, we propose to prioritize the primary instructions (PRI) for instruction scheduling. Prioritizing the primary instructions helps restrain the duplicate instruction (if on the mispredicted path) from competing with the primary instructions for computing resources. In this sense, it has the potential to improve DIE performance. When two instructions (one from the primary stream, the other from the duplicate stream) compete for an issue port, the primary instruction always wins because of its high priority assigned. The priority can be implemented by introducing an additional bit to the priority logic indicating whether it is a duplicate or not. Priority within each stream is same as in SIE using oldest-instruction first policy.

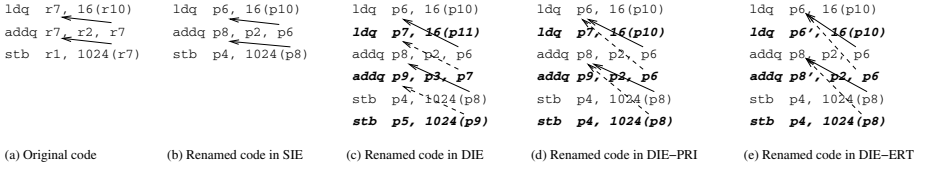


Fig. 4. A comparison of the renaming schemes in SIE, DIE, DIE-PRI, and DIE-ERT. Code in *italic* and black is the duplicate copy and arrow lines indicate the true data dependences between instructions.

To support this DIE-PRI microarchitecture, we redesign the register renaming strategy where the source operands of the duplicate are renamed to the physical registers assigned to the primary instructions producing the results and the result register renaming of the duplicate is the same as in the original DIE. Figure 4 shows an example code renamed in SIE, DIE, and the new DIE. Rename/physical registers start with "p" in the code. DIE register renaming form two redundant streams that maintain dependence chains within themselves as shown in Figure 4 (c). In the new DIE microarchitecture, although both copies are assigned physical registers if a result is to be produced, the duplicate copy forms dependences on the primary copy as illustrated in Figure 4 (d). Notice that combined with PRI, the implementation of FUS can be further simplified by eliminating the result forwarding path from floating-point ALUs to integer functional units.

4.3 Early Retiring (ERT) Redundant Instructions

Notice that the DIE performance of many low IPC applications is constrained by the instruction window size rather than the integer ALU bandwidth or issue width as discussed in the previous Section. As the size of the instruction queue

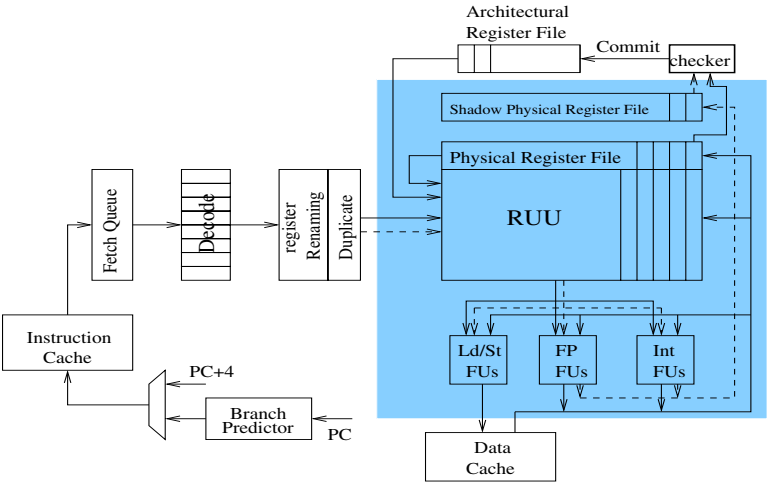


Fig. 5. DIE datapath implementing early retiring of redundant instructions (ERT)

remains fixed when entering dual-instruction execution environment, the available instruction window is effectively reduced by a factor of two. This effectively half-sized instruction window significantly handicaps the superscalar processors from extracting ILP in the aforementioned low IPC applications.

Our third proposed optimization, Early Retiring Redundant Instructions (ERT), frees up the entry of an early issued redundant instruction (either the primary or the duplicate copy) right after its issue thus to reduce the pressure of dual redundant instructions on the instruction window. Observing the inherent redundancy of the redundant copies of an instruction, it is not necessary to maintain both copies in the instruction queue once one copy is issued for execution. Only one copy is sufficient for proper commit operation. Early retiring redundant instructions (ERT) reclaims the space in the instruction window immediately after a redundant copy is issued, and guarantees that the other copy will not be early retired. In such a way, ERT can virtually increase the instruction window size to accommodate more distinct instructions for ILP extraction.

The best case of ERT is that the instruction window is filled up with instructions only with one copy and all the redundant copies are early retired, which effectively recovers the original capacity of the instruction window. To support ERT, we add a shadow physical register file with the same size of the original one. More specifically, the original physical register file functions exactly the same as in SIE and is updated only by the primary instructions. The shadow physical register file is written only by the duplicate instructions and is only for the error-checking comparison purpose at commit stage. With asymmetric renaming, source operands are supplied by the original physical register file if needed. In ERT, the register renaming is further simplified since only the primary copy needs to perform register renaming and the duplicate instruction gets the exactly same renamings as the primary. However, the result physical register

in the duplicate implicitly points to the one in the shadow register file. Figure 4 (e) shows the renamed code in DIE-ERT. The renamed result registers ***p6'*** and ***p8'*** are the shadow copy of p6 and p8. Figure 5 shows the DIE datapath supporting early retiring redundant instructions (ERT). When the instruction reaches the commit stage, it compares its results from dual execution that are stored in the same location in the physical register file and its shadow copy before updating the architectural register file. With ERT, the renamed result register number in the remaining instruction copy may suffer from soft errors before commit. This problem can be simply solved by adding a small buffer to store the result physical register number when the redundant copy is early retired. The buffer is also indexed by the register number. Thus, both the results and result register numbers are compared to verify the absence of soft errors at commit stage. A more cost-effective solution is to add a parity bit for the result physical register number. Notice that only the result register number needs to be protected. Since the physical register number only uses 7 or 8 bits (for 128 or 256 physical registers), the cost and timing overhead of this parity logic should be minimal. Note that ERT still provides full protection within the sphere of replication.

5 Experimental Results

In this section, we evaluate the effectiveness of the proposed schemes, floating-point unit sharing (FUS), prioritizing the primary instructions (PRI), and early retiring of redundant instructions (ERT) for performance improvement in DIE.

Integer ALU contention is one of the major causes resulting in the severe performance loss of high IPC applications in DIE as compared to SIE. We propose to share idle floating-point ALUs for integer ALU operations thus to reduce the competition on integer ALUs, and more importantly to provide a virtually larger integer ALU bandwidth. The performance improvement delivered by FUS (DIE-FUS) is given in Figure 6. Applying FUS, DIE-FUS recovers the performance loss of DIE to SIE by up to 58% (*gcc*) in integer benchmarks with an average of 39%, and up to 23% (*mesa*) in floating-point benchmarks with an average of 4%.

By prioritizing the primary stream during instruction scheduling, DIE-PRI helps early resolve branch instructions in the primary stream thus reducing the number of instructions on the mispredicted path in the duplicate stream that compete with the primary ones. Since most integer benchmarks are control intensive, DIE-PRI is expected to work well in these benchmarks, especially those with low branch prediction accuracy such as *perlbmk*. On the other hand, the ability of DIE-PRI to recover performance loss in DIE is quite limited for floating-point benchmarks as the control-flow instructions are much less frequent in those benchmarks. Figure 6 sees that DIE-PRI recovers the performance loss in DIE by up to 37% (*perlbmk*) in integer benchmarks with an average of 21%. DIE-PRI improves performance for all integer benchmarks. For floating-point benchmarks, DIE-PRI has a margin impact on the performance.

For applications heavily depending on a large instruction window to expose possible ILP, neither FUS nor PRI may work effectively since the performance bottleneck is not the functional unit bandwidth nor the issue width, rather is caused by the half-sized instruction window. This is especially true for many floating-point benchmarks such as *swim*, *art*, *lucas*, *mgrid*, and *applu*, whose performance can only be improved by enlarging the instruction window. Early retiring redundant instructions (ERT) virtually increases the instruction window size in DIE by immediately reclaiming the slot occupied by an early issued redundant instruction. As shown in Figure 6, DIE-ERT are very effective in boosting the performance of low IPC benchmarks. DIE-ERT reduces the performance loss of *swim* in DIE from 44% to 2% (a 96% performance recovery). On the average, DIE-ERT recovers 46% performance loss in DIE for floating-point benchmarks. For integer benchmarks, the average performance recovery delivered by DIE-ERT is around 9% with *vpr* up to 46%. However, if the benchmark suffers from frequent branch mispredictions, ERT might worsen the performance by fetching/dispatching more instructions along the wrong path that compete for resources against instructions earlier than the mispredicted branch instruction. This situation happened in *perlbmk*, instead of improving the performance, DIE-ERT causes an additional 20% performance degradation in DIE.

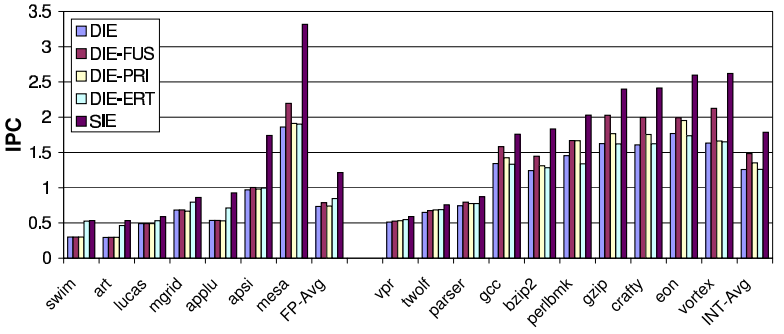


Fig. 6. Performance comparison: DIE, DIE-FUS, DIE-PRI, DIE-ERT, and SIE

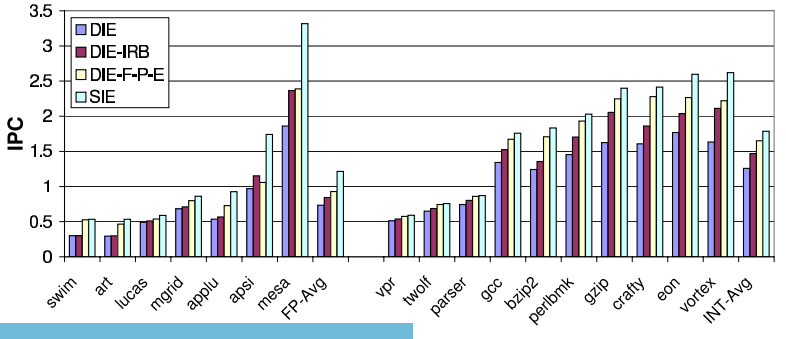


Fig. 7. Performance comparison: DIE, DIE-IRB, DIE-F-P-E, and SIE

Since the three optimizations, FUS, PRI, and ERT, have very different effects on different types of benchmarks (e.g., floating-point or integer, high IPC or low IPC, high branch prediction accuracy or low accuracy), we propose to combine them together to accommodate various of benchmarks. We also compare this new DIE-F-P-E processor (DIE augmented with FUS, PRI, and ERT) with a previous dual instruction execution instruction reuse buffer (DIE-IRB) processor [9]. The instruction reuse buffer (IRB) has the same configuration as in [9]. Our experimental results given in Figure 7 show that our combined scheme DIE-F-P-E outperforms DIE-IRB for all benchmarks except *apsi*. DIE-IRB works much better for benchmark *apsi* due to a considerable number of long latency floating-point multiplications hitting the IRB and getting the results directly from IRB. For benchmarks *swim*, *art*, *bzip2*, and *crafty*, DIE-F-P-E achieves significant performance improvement over DIE-IRB. On the average, DIE-F-P-E recovers DIE's performance loss by 78% for integer benchmarks and 54% for floating-point benchmarks, comparing to DIE-IRB's 39% recovery for integer and 15% for floating-point benchmarks, respectively. This brings us a performance loss of only 6% and 17%, an average for integer and floating-point benchmarks respectively, for providing transient error detecting dual-redundant execution in DIE-F-P-E superscalar processors (as compared to the significant performance loss, 26%/37% for integer/floating-point benchmarks in DIE).

6 Related Work

Fault detection and correction using modular redundancy has been employed as a common approach for building reliable systems. In [8], the results of the main processor and a watch-dog processor are compared to check for errors. A similar approach is presented in DIVA [2] except that the watch-dog processor is a low-performance checker processor. Cycle-by-cycle lockstepping of dual-processors and comparison of their outputs are employed in Compaq Himalaya [1] and IBM z900 [16] with G5 processors for error detection.

AR-SMT [12] and SRT [11] techniques execute two copies of the same program as two redundant threads on SMT architectures for fault detection. The slipstream processor extends the idea of AR-SMT to CMPs [18]. The SRTR work in [19] also supports recovery in addition to detection. [11] and [7] explore such redundant thread execution in terms of both single- and dual-processor simultaneous multithreaded single-chip processors.

Redundant instruction execution in superscalar processors is proposed in [6,?] for detecting transient faults. In [6], each instruction is executed twice and the results from duplicate execution are compared to verify absence of transient errors in functional unit. However, each instruction only occupies a single re-order buffer (ROB) entry. On the other hand, the dual-instruction execution scheme (SS2) in [10] physically duplicates each decoded instruction to provide a *Sphere of Replication* including instruction issue queue/ROB, functional units, physical register file, and interconnect among them. In [9], instruction reuse technique is exploited to bypass the execution stage, and thus to improve the ALU bandwidth

at the cost of introducing two comparators for source operand value comparison in the wakeup logic. A recent work [17] from the same CMU group has investigated the performance behavior of SS2 in details with respect to resources and staggered execution, and proposed SHREC microarchitecture, an asymmetric instruction-level redundant execution, to effectively reduce resource competition between the primary and redundant streams. Our work shares many of the common issues studied in previous research efforts. However, our main focus here is to exploit and maximize the utilization of the existing resources to alleviate the competition pressure on a particular part in the datapath, thus to improve the performance in redundant-instruction execution superscalar processors.

7 Concluding Remarks

This paper has investigated the performance-limiting factors in a transient-error detecting redundant-execution superscalar processor. Based on our observation that that most floating-point ALUs were idle during execution, we proposed floating-point unit sharing (FUS) to utilize these idle FP ALUs for integer ALU operations. This reduces competition for the integer ALUs significantly and increases the total available integer operation bandwidth. To further steer these available resources for useful work (instructions along the correct path), we proposed to prioritize the primary stream for instruction scheduling in DIE. This helps restrain the duplicate instructions on the mispredicted path from competing for issue slots and functional unit bandwidth, thus leading to more efficient use of these resource and performance improvement. To address the effectively half-sized instruction window in DIE, we proposed an early-retiring scheme for redundant instructions (ERT) to increase the effective size of the instruction window. By combining these three proposals, our modified processor DIE-F-P-E is able to execute instruction-level (dual) redundant code for full coverage of soft errors in the sphere of replication while experiencing a performance penalty of only 6% for integer applications and 17% for floating-point applications. As such, it recovers 78% of the 26% performance penalty normally seen in the base DIE processor for integer applications, and recovers 54% of the 37% performance penalty seen in floating-point applications. This limited performance penalty and low hardware overhead allows redundant execution, and hence fault tolerance, to be implemented on a wide range of processors at a very low cost.

References

1. Hp nonstop himalaya. http://nonstop.compaq.com/.
2. T. Austin. Diva: A reliable substrate for deep submicron microarchitecture design. In *Proc. the 32nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 196–207, November 1999.
3. D. Burger and T. M. Austin. The simplescalar tool set, version 2.0. Technical Report 1342, Computer Sciences Department, University of Wisconsin, 1997.

4. M. Gomaa, C. Scarbrough, T. Vijaykumar, and I. Pomeranz. Transient-fault recovery for chip multiprocessors. In *Proc. the International Symposium on Computer Architecture*, pages 98–109, June 2003.
5. G. Hinton, D. Sager, M. Upton, D. Boggs, D. C. n, A. Kyker, and P. Roussel. The microarchitecture of the pentium 4 processor. *Intel Technical Journal*, Q1 2001 Issue, Feb. 2001.
6. A. Mendelson and N. Suri. Designing high-performance and reliable superscalar architectures: The out of order reliable superscalar (o3rs) approach. In *Proc. of the International Conference on Dependable Systems and Networks*, June 2000.
7. S. S. Mukherjee, M. Kontz, and S. K. Reinhardt. Detailed design and evaluation of redundant multithreading alternatives. In *Proc. the 29th Annual International Symposium on Computer Architecture*, pages 99–110, May 2002.
8. M. Namjoo and E. McCluskey. Watchdog processors and detection of malfunctions at the system level. Technical Report 81-17, CRC, December 1981.
9. A. Parashar, S. Gurumurthi, and A. Sivasubramaniam. A complexity-effective approach to alu bandwidth enhancement for instruction-level temporal redundancy. In *Proc. the 31st Annual International Symposium on Computer Architecture*, June 2004.
10. J. Ray, J. Hoe, and B. Falsafi. Dual use of superscalar datapath for transient-fault detection and recovery. In *Proc. the 34th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 214–224, December 2001.
11. S. Reinhardt and S.Mukherjee. Transient fault detection via simultaneous multithreading. In *Proc. the 27th Annual International Symposium on Computer Architecture*, pages 25–36, June 2000.
12. E. Rotenberg. Ar-smt: A microarchitectural approach to fault tolerance in microprocessors. In *Proc. the International Symposium on Fault-Tolerant Computing*, pages 84–91, June 1999.
13. S. S. Sastry, S. Palacharla, and J. E. Smith. Exploiting idle floating point resources for integer execution. In *Proc. ACM SIGPLAN 1998 Conf. Programming Language Design and Implementation*, pages 118–129, June 1998.
14. T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 2002.
15. P. Shivakumar et al. Modeling the effect of technology trends on the soft error rate of combinational logic. In *Proc. International Conference on Dependable Systems and Networks*, pages 389–398, June 2002.
16. T. J. Slegel et al. IBM's S/390 G5 microprocessor design. *IEEE Micro*, 19(2):12–23, March/April 1999.
17. J. Smolens, J. Kim, J. C. Hoe, and B. Falsafi. Efficient resource sharing in concurrent error detecting superscalar microarchitecture. In *ACM/IEEE International Symposium on Microarchitecture (MICRO)*, December 2004.
18. K. Sundaramoorthy, Z. Purser, and E. Rotenburg. Slipstream processors: Improving both performance and fault tolerance. In *Proc. the 9th International Conference on Architectural Support for Programming Languages and Operating systems*, pages 257–268, 2000.
19. T. Vijaykumar, I. Pomeranz, and K. Cheng. Transient-fault recovery via simultaneous multithreading. In *Proc. the 29th Annual International Symposium on Computer Architecture*, pages 87–98, May 2002.
20. J. F. Ziegler et al. IBM experiments in soft fails in computer electronics (1978 - 1994). *IBM Journal of Research and Development*, 40(1):3–18, January 1996.

A Fault-Tolerant Routing Strategy for Fibonacci-Class Cubes

Xinhua Zhang[1] and Peter K.K. Loh[2]

[1] Department of Computer Science, National University of Singapore, Singapore
xinhua.zhang@nus.edu.sg
[2] School of Computer Engineering, Nanyang Technological University, Singapore
askkloh@ntu.edu.sg

Abstract. Fibonacci Cubes (*FCs*), together with the enhanced and extended forms, are a family of interconnection topologies formed by diluting links from binary hypercube. While they scale up more slowly, they provide more choices of network size. Despite sparser connectivity, they allow efficient emulation of many other topologies. However, there is *no* existing fault-tolerant routing strategy for *FC*s or other node/link diluted cubes. In this paper, we propose a unified fault-tolerant routing strategy for all Fibonacci-class Cubes, tolerating as many faulty components as network *node availability*. The algorithm is livelock free and generates deadlock-free routes, whose length is bounded linearly to network dimensionality. As a component, a generic approach to avoiding immediate cycles is designed which is applicable to a wide range of inter-connection networks, with computational and spatial complexity at $O(1)$ and $O(n \log n)$ respectively. Finally, the performance of the algorithm is presented and analyzed through software simulation, showing its feasibility.

1 Introduction

Fibonacci-class Cubes originate from *Fibonacci Cube* (*FC*) proposed by Hsu [1], and its extended forms are *Enhanced Fibonacci Cube* (*EFC*) by Qian [2] and *Extended Fibonacci Cube* (*XFC*) by Wu [3]. This class of interconnection network uses fewer links than the corresponding binary hypercube, with the scale increasing at $O(((1+\sqrt{3})/2)^n)$, slower than $O(2^n)$ for binary hypercube. This allows more choices of network size. In structural aspects, the two extensions virtually maintain all desirable properties of *FC* and improve it by ensuring the *Hamiltonian* property [2,3]. Besides, there is an ordered relationship of containment between the series of *XFC* and *EFC*, together with binary hypercube and regular *FC*. Lastly, they all allow efficient emulation of other topologies such as binary tree (including its variants) and binary hypercube. In essence, Fibonacci-class Cubes are superior to binary hypercube for low growth rate and sparse connectivity, with little loss of its desirable topological and functional (algorithmic) properties.

Though Fibonacci-class Cubes provide more options of incomplete hypercubes to which a faulty hypercube can be reconfigured and thus tend to find applications in fault-tolerant computing for degraded hypercubic computer systems, there are, to the best of the authors' knowledge, no existing *fault-tolerant* routing algorithms for them.

T. Srikanthan et al. (Eds.): ACSAC 2005, LNCS 3740, pp. 215 – 228, 2005.

This is also a common problem for link-diluted hypercubic variants. In this paper, we propose a unified fault-tolerant routing strategy for Fibonacci-class Cubes, named **F**ault-**T**olerant **F**ibonacci **R**outing (**FTFR**), with following strengths:

- It is applicable to all Fibonacci-class Cubes in a unified fashion, with only minimal modification of structural representation.
- The maximum number of faulty components tolerable is the network's node availability [4] (the maximum number of faulty neighbors of a node that can be tolerated without disconnecting the node from the network).
- For a n-dimensional Fibonacci-class Cube, each node of degree deg maintains and updates at most $(deg + 2)\cdot n$ bits' vector information, among which: 1) a n-bit availability vector indicating the local non-faulty links, 2) a n-bit input link vector indicating the input message link, 3) all neighbors' n-bit availability vector, indicating their link availability.
- Provided the number of component faults in the network does not exceed the network's node availability, and the source and destination nodes are not faulty, FTFR guarantees that a message path length does not exceed $n+H$ empirically and $2n+H$ theoretically, where n is the dimensionality of the network and H is the *Hamming* distance between source and destination.
- Generates deadlock-free and livelock-free routes.
- Can be implemented almost entirely with simple and practical routing hardware requiring minimal processor control.

The rest of this paper is organized as follows. Section 2 reviews several versions of definitions of Fibonacci-class Cube, together with initial analysis on their properties. Section 3 presents a Generic Approach to Cycle-free Routing (**GACR**), which is used as a component of the whole strategy. Section 4 elaborates on the algorithm FTFR. The design of a simulator and simulation results will be presented in section 5. Finally, the paper is concluded in section 6.

2 Preliminaries

2.1 Definitions

Though Fibonacci-class Cubes are very similar and are all based on a sequence with specific initial conditions, they do have some different properties that require special attention. The well-known *Fibonacci number* is defined by: $f_0 = 0$, $f_1 = 1$, $f_n = f_{n-1} + f_{n-2}$ for $n \geq 2$. In [1], the *order-n Fibonacci code* of integer $i \in [0, f_n - 1]$ $(n \geq 3)$ is defined as $(b_{n-1}, \cdots, b_3, b_2)_F$ where b_j is either 0 or 1 for $2 \leq j \leq (n-1)$ and $i = \sum_{j=2}^{n-1} b_j f_j$. Then *Fibonacci Cube of order n* ($n \geq 3$) is a graph $FC_n = < V(f_n), E(f_n) >$, where $V(f_n) = \{0, 1, \cdots, f_n - 1\}$ and $(i, j) \in E(f_n)$ if and only if the *Hamming* distance between I_F and J_F is 1, where I_F, J_F stand for the Fibonacci codes of i and j, respectively. An equivalent definition is: let $FC_n = (V_n, E_n)$, then

$V_3 = \{1,\ 0\}$, $V_4 = \{01,\ 00,\ 10\}$ and $V_n = 0 \| V_{n-1} \bigcup 10 \| V_{n-2}$ for $n \geq 5$, where $\|$ denotes the concatenation operation and has higher precedence than union operation $\bigcup$. Two nodes in FC_n are connected by an edge in E_n if and only if their *Hamming* distance is 1. To facilitate a unified discussion of Fibonacci-class Cubes, V_n can be a further defined in an obviously equivalent form: a set of all possible $(n-2)$ -bit binary numbers, none of which has two consecutive 1's. This definition is important for the discussion of properties in the next sub-section.

Enhanced Fibonacci Cube and *Extended Fibonacci Cube* can be defined in a similar way. Let $EFC_n = < V_n, E_n >$ denote the *Enhanced Fibonacci Cube of order n* ($n \geq 3$), then $V_3 = \{1,0\}$, $V_4 = \{01,\ 00,\ 10\}$, $V_5 = \{001,\ 101,\ 100,\ 000,\ 010\}$, $V_6 = \{0001,\ 0101,\ 0100,\ 0000,\ 0010,\ 1010,\ 1000,\ 1001\}$ and $V_n = 00 \| V_{n-2} \bigcup 10 \| V_{n-2} \bigcup 0100 \| V_{n-4} \bigcup 0101 \| V_{n-4}$ for $n \geq 7$. Two nodes in EFC_n are connected by an edge in E_n if and only if their labels differ in exactly one bit position, i.e., *Hamming* distance is 1.

A series of *Extended Fibonacci Cubes* is defined as $\{XFC_k(n) \mid k \geq 1,\ n \geq k+2\}$, where $XFC_k(n) = \{V_k(n), E_k(n)\}$, $V_k(n) = 0 \| V_k(n-1) \bigcup 10 \| V_k(n-2)$ for $n \geq k+4$. As initial conditions for recursion, $V_k(k+2) = \{0,\ 1\}^k$ meaning the *Cartesian* product of k sets of $\{0, 1\}$, and $V_k(k+3) = \{0,\ 1\}^{k+1}$. Two nodes in $XFC_k(n)$ are connected by an edge in $E_k(n)$ if and only if their *Hamming* distance is 1.

2.2 Properties of Fibonacci-Class Cubes

In this section, we introduce an important property for our algorithm. Let current node address be *u* and destination node address be *d*, then each dimension corresponding to 1 in $u \oplus d$ is called *preferred dimension*, where $\oplus$ stands for bitwise XOR operation. Other dimensions are called *spare dimension*. As Fibonacci-class Cubes are defined by link dilution, it is likely that links in some preferred dimensions are not available at a packet's current node. But the following proposition guarantees that in a fault-free setting, there is always at least one preferred dimension available at the present node. Unlike in binary hypercube, this is not a trivial result.

(*Proposition* 1) In a fault-free *FC*, *EFC* or *XFC*, there is always a preferred dimension available at the packet's present node before the destination is reached.

Proof : Consider an *n*-dimensional Fibonacci-class Cube, which means *FC, XFC* and *EFC* of order *n* + 2. Let the binary address of current node be $a_{n-1} \cdots a_1 a_0$ and the destination be $d_{n-1} \cdots d_1 d_0$. Let the rightmost (least significant) bit correspond to dimension 0 while the leftmost bit correspond to dimension $n-1$.

Case I: *Fibonacci Cube* FC_{n+2}. Obviously, if the destination has not been reached, there is always a preferred dimension $i \in [0, n-1]$. If $a_i = 1$ and $d_i = 0$, then there is always a preferred link available at dimension *i* because changing one '1' in a valid address into 0 always produces a new valid address. So we only need to consider $a_i = 0$ and $d_i = 1$. When $n \leq 3$, the proposition can be easily proven by enumeration.

So now we suppose $n \geq 4$.

1) If $i \in [1, n-2]$, then $d_{i-1} = 0$, $d_{i+1} = 0$. If $a_{i-1} = 1$, then $i - 1$ is an available preferred dimension. If $a_{i+1} = 1$, then $i+1$ is an available preferred dimension. If $a_{i-1} = a_{i+1} = 0$, then dimension i is an available preferred dimension because inverting a_i to 1 will not produce two consecutive 0's in the new nodes address.

2) If $i = 0$, then $d_1 = 0$. If $a_1 = 1$, dimension 1 is an available preferred dimension. If $a_1 = 0$, then dimension 0 is an available preferred dimension because setting a_0 to 1 will give a valid address.

3) If $i = n - 1$, then $d_{n-2} = 0$. If $a_{n-2} = 1$, then dimension $n - 2$ is an available preferred dimension. If $a_{n-2} = 0$, then dimension $n-1$ is an available preferred dimension.

Case II: $XFC_k(n+2)$. Suppose there is a preferred dimension i. If $i < k$, then inverting a_i will always produce a valid address. If $i \geq k$, the discussion is the same as case I.

Case III: EFC_{n+2}. The discussion is similar to case I, but much more complicated. Basically, we just need to discuss over the leftmost preferred dimension. Detailed proof is omitted here due to limited space, but is available at http://www.comp.nus.edu.sg/~zhangxi2/proof.pdf.

The proposition implies that whenever a spare dimension is used, either a faulty component is encountered or all neighbors on preferred dimensions have been visited previously. For the latter case, all such preferred dimensions must have been used as spare dimensions sometime before. So both cases can be boiled down to the encounter of faulty components (now or before). It is also implied that certain algorithm can be applied to all the three types of network and the existence of at least one preferred dimension is guaranteed in a certain sense, given the number of faulty components does not exceed node availability. This is the underlying idea of FTFR.

3 Generic Approach to Cycle-Free Routing (GACR)

As a component of FTFR, we digress from the mainframe for one section and propose now a generic approach to avoiding cycles in routing by checking the traversal history. The algorithm takes only $O(1)$ time to update the coded history record and $O(n)$ time to check whether a neighbor has been visited before (can be virtually reduced to $O(1)$ by utilizing parallelism). Other advantages include its wide applicability and easy hardware implementation. It applies to all networks in which links only connect node pairs whose *Hamming* distance is 1 (called *Hamming* link). All networks constructed by node or link dilution meet this criterion. An extended form of the algorithm can be applied to those networks with $O(1)$ types of non-*Hamming* links at each node. Thus, such networks as *Folded Hypercube*, *Enhanced Hypercube* and *Josephus Cube* [5] can also use this algorithm.

3.1 Basic GACR

The traversal history is effectively an ordered sequence of dimensions used when leaving each visited node. We use Figure 1 for illustration.

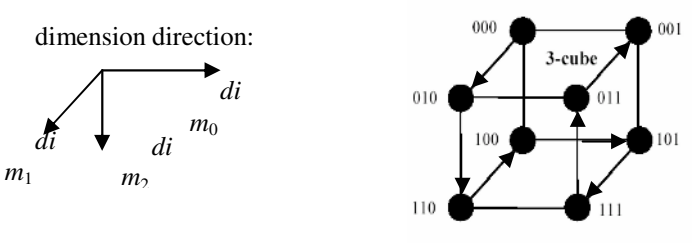


Fig. 1. Basic GACR illustration

The route that originates from 000 can be recorded as: 1210121. An obvious equivalent condition for a route to contain a cycle is: there exists a way of inserting '(' and ')' into the sequence such that each number in the parenthesis appears for an even number of times. For example, in 1(21012), 0 appears only one time. In (1210121), 1 and 2 appear for an even time but 0 still appears for an odd number of times. So neither case forms a cycle. But for a sequence of 1234243, there must be a cycle: 1(234243). Suppose at node *p*, the history sequence is $a_1 a_2 \cdots a_n$, and it is guaranteed that no cycle exists hitherto, then to check whether using dimension a_{n+1} will cause any cycle, we only need to check whether in $(a_n a_{n+1})$, $(a_{n-2} a_{n-1} a_n a_{n+1})$, $(a_{n-4} a_{n-3} a_{n-2} a_{n-1} a_n a_{n+1}) \ldots$ each number will appear for an even number of time.

We first introduce the basic form of this algorithm that applies only to networks constructed by node/link dilution from binary hypercube. This algorithm is run at each intermediate node to ensure that the next node has never been visited before.

(*Algorithm* 1) *Basic GACR*

The data structure is a simple array: port[], with each element composed of $\lceil \log n \rceil$ bits. port[*i*] records the port used when exiting the node that the packet visited $i+1$ hops before. So when leaving a node, we only need to append the exiting dimension to the *head* of the array port[]. As each node has only *n* ports and each dimension *c* at node *a* has the same meaning at node *b*, only $\lceil \log n \rceil$ bits are needed to represent these *n* possibilities. At the source node, port[ ] is null. Suppose at node *x*, the length of the array is *L*. After running the following code segment, each 0 in *mask* corresponds to a dimension, the using of which will cause an immediate cycle.

```
unsigned GACR ( unsigned port[], int L) {
  unsigned mask = 0, history = 1 << port[0];
  for (int k = 1; k < L; k ++)        {
     // invert the bit of history corresponding to dimension 'port[k]'
     history = history XOR (1 << port[k]);
     if (k is an odd number and history has a single 1)
          mask = mask OR history;
  }
    return ~mask;                          // bitwise NOT of mask
}
```

For instance, given the dimension sequence 875865632434121 from source to present, the mask should be 000010011, because in 875865632434121*a*, a cycle is formed when *a* = 2, 3, 5, 6, 7, or 8. All operations in this algorithm are basic logic operations. The logic check for whether *history* $x_{n-1}\cdots x_0$ has a single 1 is $\sum_{i=0}^{n-1} x_{n-1}\cdots x_{i+1}\overline{x}_i x_{i-1}\cdots x_0$ (logic sum of bit products) where $\overline{x}_i$ is the complementary of x_i. The check can be implemented to cost only one clock cycle, with *n* AND gates and 1 OR gate. But in software simulation, it takes $O(n)$ time, which requires attention during simulation. This algorithm also has the strength that the time cost can be reduced to virtually zero because it can be executed when the packet is still queuing in the buffer, making parallelism and pipelining possible. Though the time complexity is $O(L_{max})$ and message overhead is $O(L_{max}\log n)$, where L_{max} is the length of the longest path that the packet can traverse, in most routing algorithms, $L_{max} = O(n)$ thus $O(L_{max}\log n) = O(n\log n)$. So both time and spatial complexity are within the acceptable bound in practice. For example, the technique used by [6] incurs message overhead of $(n+1)\lceil \log_2 n \rceil$ bits for *n*-dimensional binary hypercube.

3.2 Extended GACR

If the network has $O(1)$ number of non-*Hamming* links at each node and those links can be represented in a common way, then basic GACR can be easily extended. For example, in *Josephus Cube* $JC(n)$ [5], we denote the complementary link (between $x_{n-1}x_{n-2}\cdots x_0$ and $\overline{x}_{n-1}\overline{x}_{n-2}\cdots\overline{x}_0$,) as dimension *n* and the *Josephus* link (between $x_{n-1}\cdots x_2x_1x_0$ and $x_{n-1}\cdots x_2\overline{x}_1\overline{x}_0$) as dimension $n+1$. Then the basic GACR can be extended as follows. *mask2* = 0 and *mask3* = 0 represent that using complementary link and *Josephus* link will result in a cycle, respectively.

(*Algorithm* 2) *Extended GACR*

```
void ExGACR (unsigned port[], int L, unsigned &mask1,
                 unsigned &mask2, unsigned &mask3) {
  // mask1 to mask3 are called by reference, mask1 is the same as mask in Algo. 1
  unsigned dim, history = 1 << port[0];
  mask1= 0;      mask2 = mask3 = 1;

  for (int k = 1; k < L; k ++)          {
      if ( port[k] < n )              // once exit through Hamming links
          dim = 1 << port[k];
      else if ( port[k] == n ) // once exit through complementary link
          dim = (1<<n) - 1;
      else dim = (unsigned) 3;        // once exit through Josephus link
      history = history XOR dim;
      if (history has a single 1)
                                  // now when k is even, we also have to check
          mask1 = mask1 OR history;  // for cycles caused by Hamming link
```

```
        else if (history is straight 1)
                // check if history has straight 1's for the cycle caused by complementary
            link
            mask2 = 0;
        else if (history = = (unsigned) 3)
            // check the rightmost two bits for the cycles caused by Josephus link
            mask3 = 0;
    }
    mask1 = ~ mask1;
}
```

4 Fault-Tolerant Fibonacci Routing (FTFR)

4.1 Definition and Notation

Now we will go back to the mainframe of the routing strategy. In a Fibonacci-class Cube of order $n + 2$ (n-dimensional), each node's address is a n-bit binary number. Let the source node s be $(a_{n-1} \cdots a_1 a_0) \in \{0, 1\}^n$ and the destination node d be $(d_{n-1} \cdots d_1 d_0) \in \{0, 1\}^n$. Denote the neighbor of node u along the i^{th} dimension $(0 \le i < n)$ as $u^{(i)}$, by inverting the i^{th} bit of u's binary address. For convenience we define following terms.

1) *input link vector* $I(x)$. An n-bit *input link vector* at node x is defined as $I(x) = [l_{n-1} \ldots l_1 l_0]$, where $l_i = 0$ if the message goes to the current node along the dimension i link ($0 \le i < n$). $l_j = 1$ for $j \ne i$. Setting the corresponding bit to 0 for a used input link prevents the link from being used again immediately for message transmission, causing the message to "oscillate" back and forth. An input link vector has straight 1's for a new message.

2) *availability vector* $AV(x)$. At each node x, the n-bit binary number *availability vector* $AV(x)$ records a bit string, indicating by '1' what dimensions are available at x, and by '0' what dimensions are unavailable. Here a dimension i is available if there is a non-faulty link at x to $x^{(i)}$. For example, in Figure 2, node 1001 and link (0000, 0001) are faulty. The availability vector for all nodes is listed in Table 1.

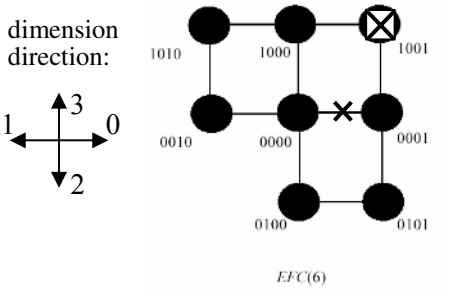


Fig. 2. *Availability vector* example

Table 1. *Availability vector* for Fig. 2

node	*AV*
0000	1110
0001	0100
0010	1010
0100	0101

nod	*AV*
010	0101
100	1010
100	0000
101	1010

Availability vector is crucial for the generic applicability of the routing algorithm to all Fibonacci-class Cubes. It is in essence a distributed representation of network topology and fault distribution.

3) *mask vector DT*. To prevent cycles in the message path actively (not only by checking the immediate cycle at each step), a *mask vector* is introduced as part of the message overhead, defined as $DT = [t_{n-1} \cdots t_1 t_0]$. At source node, *DT* is cleared to be straight 1. After that, whenever a spare dimension is used, the corresponding bit in *DT* is set to 0 and cannot be set back to 1. Spare dimensions whose corresponding bit in *DT* is 0 cannot be used, i.e., dimensions that are spare at the source node can be used for at most two times. But different from many existing algorithms (e.g. [7]), each dimension which is preferred at the source can be used more than once. When it is used for the first time, *DT* doesn't record it. But when it is used as a spare dimension later, the corresponding bit in *DT* is masked, so that it can't be used as a spare dimension again. Finally, each node periodically exchanges its own availability vector with all neighbors.

4.2 Description of FTFR

Empirically, the number of faults FTFR can tolerate is the network's node availability, a value constant for a network which is independent of each node. There is an intricate mechanism for choosing candidate dimension when more than one preferred dimension are available, or when no preferred but several spare dimensions are available. First of all, the *GACR* is applied to generate a mask *M*. Only those dimensions whose corresponding bit in (*M* AND $I(x)$ AND $AV(x)$) is 1 are further investigated. These dimensions are called **adoptable**. For illustration, we use the following Figure 3, in which '*s*' and '*p*' stand for spare and preferred dimension, respectively. We divide our discussion into two cases.

(**Case I**) If there are several adoptable *preferred* neighbors (like A and B), we choose the one which has the largest number of non-faulty preferred dimensions. To break tie, we compare their number of non-faulty spare dimension. If still tie, choose the lowest dimension. Equivalently, if the dimensionality of the network is *n*, then the score to compare is $n \cdot N_p + N_s$, where N_p and N_s stand for the number of preferred and spare dimensions respectively. Here, for A, $n \cdot N_p + N_s = 2n + 2$, while for B, the score is $n + 3$. So A is chosen.

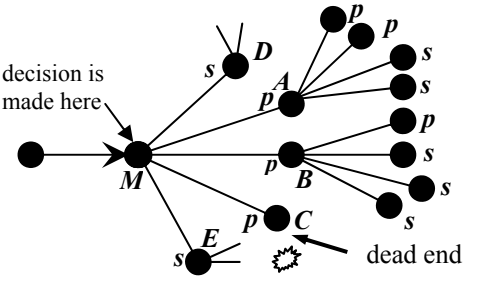


Fig. 3. Illustration of *FTFR*

(**Case II**) If at current node M, there are no adoptable preferred dimensions, spare dimensions have to be used, like D and E. Firstly, the eligibility is checked by *DT*. Then just like in case I, we compare $n \cdot N_p + N_s$. After one spare dimension is finally chosen, its corresponding bit in *DT* is masked to 0, so that it will not be used as spare dimension again.

The $m = n \cdot N_p + N_s$ is a heuristic score. After extensive experiment, it is found that minor modifications can be made to *m* so as to improve the performance of FTFR. Suppose the dimension under consideration is *i* and inverting the i^{th} bit of *destination d* produces $d^{(i)}$. If $d^{(i)}$ is a valid node address in that Fibonacci-class Cube, attaching some priority to dimension *i* will be helpful in reducing the number of hops. Hence, we add the value of network node availability to *m* for that candidate dimension in such a case.

The following is the pseudo-code for the two core routing functions. *GetNext* is run at *M*, which looks ahead at A, B, C, D and E. Vector *adoptable* = *AV*(*M*) AND *I*(*M*) AND (*GACR* result).

```
unsigned GetNext (unsigned current, unsigned destination,
                  unsigned adoptable, unsigned *DT) {
int max1, max2;

If (current = = destination)
  return REACH_DESTINATION;
If exists adoptable 1->0 preferred dimension
  mask = source & ~destination & adoptable;
  //choose the dimension which has the largest n*Np+Ns. Record the value in max1
  call OneBest(current, destination, mask, *DT, &max1)
If exists adoptable 0->1 preferred dimension
  mask = ~source & destination & adoptable;
  //choose the dimension which has the largest n*Np+Ns. Record the value in max2
  call OneBest(current, destination, mask, *DT, &max2)
If either case above gives a valid dimension
  return the dimension corresponding to the larger of max1
      and max2. If tie, use 1->0 link.
If there is an adoptable 0->0 spare dimension
  mask = ~current & ~destination & adoptable & *DT;
  call OneBest(current, destination, mask, *DT, &max1)
If there is an adoptable 1->1 spare dimension
  mask = current & destination & adoptable & *DT;
  call OneBest(current, destination, mask, *DT, &max2)
   If either case above gives a valid dimension
  update DT;
  return the dimension corresponding to the larger of max1
      and max2. If tie, use 1->1 link.
   Otherwise
  return ABORT;                                  // should not reach here
}
```

The following function *OneBest* is run for "scoring" neighbors, A, B, C,…. and returns the selected dimension. Each 1 in *cand* corresponds to a candidate dimension

waiting to be considered. *max* records the largest $n \cdot N_p + N_s$. If all candidates have no adoptable outlets, *max* is set to –1 (unchanged as before *OneBest* is called) and return NOT_FOUND.

```
unsigned OneBest(unsigned current, unsigned destination, unsigned cand, unsigned DT, int *max) {
  int total, prefer, spare, best, d;   *max←−1;
  for(d = 0; d < n; d ++)   {
      If (d is a candidate dimension in cand)   {
            unsigned neighbor = current node's neighbor along dimension d
            If (neighbor = = destination)    {
                *max = INFINITY;
                return d;
            }
            prefer  = number of adoptable preferred dimension at neighbor
            spare  = number of adoptable spare dimension at neighbor
                        (note DT is used here)
            total = n ×prefer + spare;
            If destination has link on dimension d
                              in a (imagined) fault-free
                           setting
                total = total + Node_Availability;
            If (total > *max)  {
                *max = total;
                best = d;                          // record current winner
            }
      }
  }
  If (*max = −1)                                   // no qualified dimension is found
       return NOT_FOUND;
  return best;
  }
```

It is obvious that the algorithm possesses all the properties listed in Section 1. The only uncertain thing, which is why we call it a heuristic algorithm, is that no guarantee can be made that FTFR will never fail to find a route (*GetNext* may return ABORT) when there really exists one. We call it false abortion. We enumerated all possible locations of faulty components and (source, destination) pairs for three kinds of Fibonacci-class Cubes with dimensionality below 7 and no false abortion occurs. For higher dimensional cases, we can only randomly set faults and pick (source, destination) pairs. After one month's simulation on a 2.3 GHz CPU, still no false abortion is detected. In the next section, we will test the performance of the FTFR extensively.

For example, suppose in a 9-dimensional Regular Fibonacci Cube FC_{11}, there are two faulty nodes 000001000 and 000000001 while no link is faulty. Source is 101010100 and destination is 000001001. At the beginning, there are 4 adoptable $1 \rightarrow 0$ preferred dimensions, namely 2, 4, 6, 8, whose $n \cdot N_p + N_s$ scores are $4\times9+1$, $4\times9+2$, $4\times9+0$, $4\times9+0$ respectively. After updating for dimensional availability at

destination, their final scores are 37, 38, 39, and 39, respectively (node availability is 3). The $0 \rightarrow 1$ preferred dimension 0 has score 39. Thus dimension 6 or 8 can be chosen and we use the smaller one. After using dimension 6 8 0 4 0 2, the packet reaches 000000000, where neither of the two preferred dimensions (3 and 0) is adoptable because they both lead to a faulty node. So a spare dimension has to be used. The input dimension is 2 and using dimension 4 will lead to cycle. Therefore, there are only 5 adoptable dimensions, namely 1, 5, 6, 7, 8. Their scores are: 14, 25, 25, 25, and 27, respectively. So spare dimension 8 is chosen and its corresponding bit in *DT* is masked.

5 Simulation Results

A software simulator is constructed to imitate the behavior of the real network [8, 9], and thus test the performance of our algorithm. The assumptions are: (1) fixed packet-sized messages, (2) source and destination nodes must be non-faulty, (3) eager readership is employed when packet service rate is faster than packet arrival rate, (4) a node is faulty when all of its incident links are faulty, (5) a node knows status of its links to its neighboring nodes, (6) all faults are fail-stop, (7) location of faults, source and destination are all randomly chosen by uniform distribution.

The performance of the routing algorithms is measured by two metrics: *average latency* and *throughput*. Average latency is defined as *LP*/*DP*, where *LP* is the total latency of all packets that have reached destination while *DP* is the number of those packets. Throughput is defined as *DP*/*PT*, where *PT* is the total routing process time taken by all nodes.

5.1 Comparison of FTFR's Performance on Various Network Sizes

In this section, FTFR is applied to fault-free *FC*, *EFC*, XFC_1 and binary hypercube. Their average latency and throughput are shown in Figure 4 and 5 respectively.

In Figure 4, it can be observed that the average latency of regular/Enhanced/Extended Fibonacci Cubes increases when the networks dimension *n* is below 19. As the network size grows, its diameter increases and a packet has to take a longer path to reach its destination, resulting in a higher average latency. The *EFC* has the highest latency among three because when dimension is large enough, the number

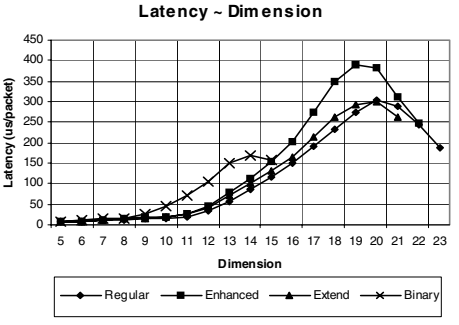


Fig. 4. Latency of Fault-free Fibonacci-Class Cubes

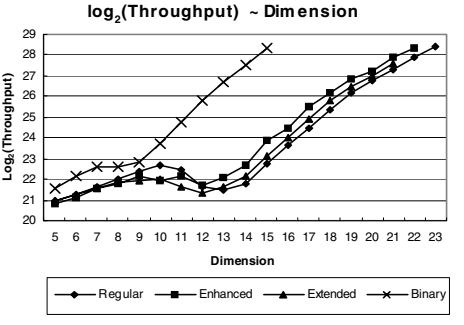


Fig. 5. Throughput (logarithm) of Fault-free Fibonacci-class Cubes

of nodes in *EFC* is the largest among Fibonacci-class Cubes of the same dimension. After the dimension reaches 19 or 20, the latency decreases. This is because the scale of the network becomes so large that the simulation time is insufficient to saturate the network with packets, i.e., the number of packets reaching destination is lower than the total allowable packet number. So the packets in these networks spend less time waiting in output queue or injection queue, while that portion of time comprises a major part of latency for lower dimensional networks that get saturated with packets during the simulation. As n = 19~20 is already adequate for demonstrating the performance of FTFR, we do not wait for network saturation. Binary hypercube, a special type of *XFC*, shows a similar trend, with latency starting to decrease at n = 15. This also goes well with the fact that the number of nodes in Fibonacci-class Cube and binary hypercube are $O((1+\sqrt{3})^n/2^n)$ and $O(2^n)$ respectively. $(1+\sqrt{3})/2:2 \approx 15{:}20$. Due to limited physical memory, simulation for binary hypercube is conducted up to n = 15.

In Figure 5, it is demonstrated that the throughput of all networks is increasing as the dimension increases from 12 to over 20. This is due to the parallelism of the networks and the increase in the number of nodes, which can generate and route packets in the network, is faster as compared with the FTFR time complexity $O(n \log n)$. By increasing the network size, the number of links is also increasing at a higher rate than the node number. This in turn increases the total allowable packets in the network. With parallelism, more packets will be delivered in a given duration. For the same reason mentioned in the previous discussion of latency, *EFC* has the largest throughput among the three types of Fibonacci-class Cube. An interesting observation is that for dimensions between 11~13, the throughput decreases and increases again afterwards. One possible explanation is: the complexity of FTFR is $O(n \log n)$. For large n, the variation in $\log n$ is small compared with small n. Thus the difference given by $\log n$ will be small and the trend of throughput is similar to an $O(n)$ routing algorithm. For small n, however, the contribution of $\log n$ is comparable with the increase rate of networks size, which leads to the seemingly irregularity. On the other hand, when dimension is small (below 11), the network is too small to display this characteristic. For Fibonacci-class Cube, the irregular interval is 11~13, while for binary hypercube, such an interval is 8~9. This again tallies with $(1+\sqrt{3})/2:2 \approx 8.5{:}12$.

5.2 Comparison of FTFR's Performance on Various Numbers of Faults

In this sub-section, the performance of FTFR is measured by the varying the number of faulty components in network. The result for $XFC_{13}(16)$ is shown in Figure 6.

It is clear that when the number of faults increases, the trend of average latency is to increase while the throughput is to decrease. This is because when more faults appear, the packet is more likely to use spare dimensions which make the final route longer. In consequence, the latency increases and throughput decreases. However, there are some special situations when the existence of faults reduces the number of alternative output ports available, and thus accelerates the routing process. The varying number of faults has more evident influence when the network size is small. With fixed number of faults, there are fewer paths available for routing in smaller networks than in larger ones. Thus making some of the paths unavailable in smaller networks will bring about more significant influence. While the networks grows larger and larger, the total number of

nodes in n-dimension network is $O((1+\sqrt{3})^n/2^n)$ and maximum faulty component number tolerable is $O(n)$. So the influence of faults will bring about less and less influence on the overall statistical performance on the network. That explains why the throughput and latency fluctuate in Fig. 6. However, the overall trend is still correct. On the other hand, as the number of faults tolerable in Fibonacci-class Cubes of order n is approximately $\lfloor n/3 \rfloor$ or $\lfloor n/4 \rfloor$ [1,2,3], we have to use networks of large dimension to provide a large enough number of faults for comparison. Figure 7, 8, and 9 present the influence of varying number of faults for 20-dimensional regular Fibonacci Cube, 19-dimensional *EFC*, and 18-dimensional *XFC* respectively.

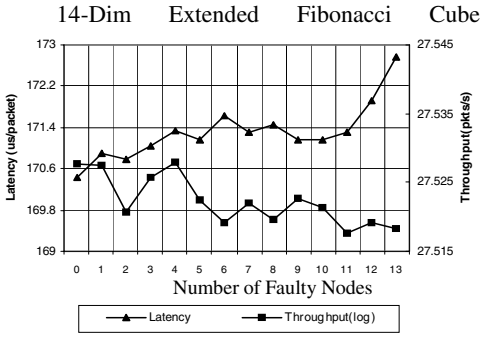


Fig. 6. Latency and Throughput (logarithm) of a faulty 14-dim Extended Fibonacci Cube

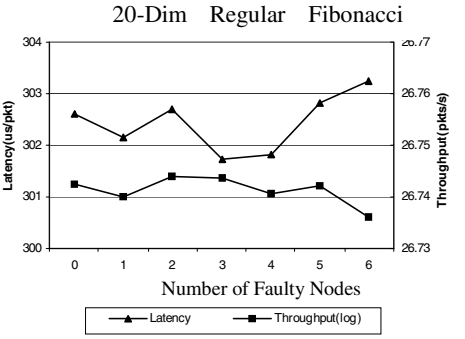


Fig. 7. Latency and Throughput (logarithm) of a faulty 20-Dim FC

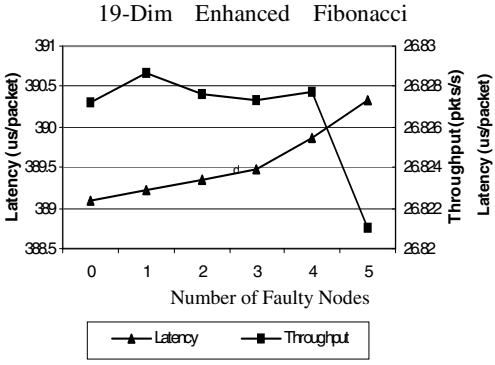


Fig. 8. Latency and Throughput (logarithm) of a faulty 19-Dim *EFC*

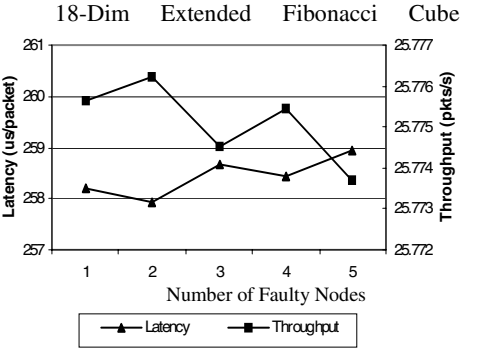


Fig. 9. Latency and Throughput (logarithm) of a faulty 18-Dim *XFC*

The fluctuation of the result needs to be examined carefully. For example, the latency in Figure 9 varies only in the range of below 1%. For different runs of simulation, the faulty components' location is randomly distributed. Similarly, messages have different destinations according to uniform distribution. If we examine the standard deviation of the result (not shown), it is observed that such a small

variation as in Figure 9 is within the 95% confidence interval for almost all situations. Thus, it is more reasonable to focus on the trend of the statistical results, rather than the exact value.

6 Conclusion

In this paper, a new effective fault-tolerant routing strategy is proposed for Fibonacci-class Cubes in a unified fashion. It is livelock free and generates deadlock-free routes with the path length strictly bounded. The space and computation complexity as well as message overhead size are all moderate. Although the Fibonacci-class Cubes may be very sparsely connected, the algorithm can tolerate as many faulty components as the network node availability. The component mechanism which ensures cycle-freeness is also generically applicable to a wide range of network topologies. Simulation results show that the algorithm scales well with network size and is empirically immune to false abortion. Future work needs to be done on further increasing the number of faulty components tolerable, possibly by careful categorization of faults based on their specific location as in [8]. Physical implementation, such as *Field Programmable Gate Array*, can also be done to evaluate the algorithm's efficiency and feasibility more accurately.

References

1. Hsu, W. J. (1993). Fibonacci Cubes-A New Interconnection Topology. *IEEE Transactions on Parallel and Distributed Systems* 4[1], 3-12.
2. Qian, H. & Hsu, W. J. (1996). Enhanced Fibonacci Cubes. *The Computer Journal* 39[4], 331-345.
3. Wu, J. (1997). Extended Fibonacci Cubes. *IEEE Transactions on Parallel and Distributed Systems* 8[12], 1203-1210.
4. Fu, A. W. & Chau, S. (1998). Cyclic-Cubes: A New Family of Interconnection Networks of Even Fixed-Degrees. *IEEE Transactions on Parallel and Distributed Systems* 9[12], 1253-1268.
5. Loh, P. K. K. & Hsu, W. J. (1999). The Josephus Cube: A Novel Interconnection Network. *Journal of Parallel Computing* 26, 427-453.
6. Chen, M.-S. & Shin, K. G. (1990). Depth-First Search Approach for Fault-Tolerant Routing in Hypercube Multicomputers. *IEEE Transactions on Parallel and Distributed Systems* 1[2], 152-159.
7. Lan, Y. (1995). An Adaptive Fault-Tolerant Routing Algorithm for Hypercube Multicomputers. *IEEE Transactions on Parallel and Distributed Systems* 6[11], 1147-1152.
8. Loh, P. K. K. & Zhang, X. (2003). A fault-tolerant routing strategy for Gaussian cube using Gaussian tree. *2003 International Conference on Parallel Processing Workshops*, 305-312.
9. Zhang, Xinhua (2003). Analysis of Fuzzy-Nero Network Communications. *Undergraduate Final Year Project*, Nanyang Technological University.

Embedding of Cycles in the Faulty Hypercube

Sun-Yuan Hsieh

Department of Computer Science and Information Engineering,
National Cheng Kung University, No. 1, Ta-Hsueh Road,
Tainan 701, Taiwan
hsiehsy@mail.ncku.edu.tw

Abstract. Let f_v (respectively, f_e) denote the number of faulty vertices (respectively, edges) in an n-dimensional hypercube. In this paper, we show that a fault-free cycle of length of at least $2^n - 2f_v$ can be embedded in an n-dimensional hypercube with $f_e \leq n-2$ and $f_v + f_e \leq 2n-4$. Our result not only improves the previously best known result of Sengupta (1998) where $f_v > 0$ or $f_e \leq n-2$ and $f_v + f_e \leq n-1$ were assumed, but also extends the result of Fu (2003) where only the faulty vertices are considered.

Keywords: Hypercubes; Interconnection networks; Fault-tolerant embedding; Cycle embedding.

1 Introduction

It is well known that the hypercube is one of the most versatile and efficient architecture yet discovered for building massively parallel or distributed systems. It possesses quite a few excellent properties such as recursive structure, regularity, symmetry, small diameter, relatively short mean internode distance, low degree, and much small link complexity, which are very important for designing massively parallel or distributed systems [12].

An *embedding* of one *guest graph* G into another *host graph* H is a one-to-one mapping f from the vertex set of G into the vertex set of H [12]. An edge of G corresponds to a path of H under f. An embedding strategy provides us a scheme to emulate a guest graph on a host graph. Therefore, those algorithms developed in a guest graph can be executed well on a host graph.

Since vertex faults and edge faults may happen when a network is put in use, it is practically meaningful to consider faulty networks. The problem of fault-tolerant embedding has received much attention recently [3,4,5,7,8,9,10,11,13,15, 17,18,19,20,22,23].It can be briefly stated as follows: how to embed a (fault-free) guest graph of as large as possible into a given faulty host graph. In this paper, we have specialized the problem with cycle being the guest graph and the hypercube being the host graph. For other guest graphs and host graphs, the interested readers may consult [3,5,9,10,13,19,20,22,23].

A *Hamiltonian cycle* in a graph G is a cycle that contains every vertex exactly once. A graph G is said to *Hamiltonian* if it contains a Hamiltonian cycle. It is well known that the hypercube is Hamiltonian. Clearly, if the hypercube contains

T. Srikanthan et al. (Eds.): ACSAC 2005, LNCS 3740, pp. 229–235, 2005.

faulty vertices or faulty edges, then it may not contain a Hamiltonian cycle. Our goal aims at finding a fault-free cycle as large as possible in a faulty hypercube. Many results regarding fault-tolerant cycle embedding in a hypercube with only faulty vertices or only faulty edges or both faulty vertices and edges have been proposed in the literature [2,4,11,15,18,19,23].

Let f_v (respectively, f_e) be the number of faulty vertices (respectively, edges) in an n-dimensional hypercube Q_n. In case of considering only faulty edges, Alspach et al.[2] showed that Q_n contains $\lfloor n/2 \rfloor$ edge-disjoint Hamiltonian cycles. This implies that at least one Hamiltonian cycle can be determined with $f_e \leq \lfloor n/2 \rfloor - 1$. Later, it was shown that a fault-free Hamiltonian cycle can be embedded in Q_n with $f_e \leq n-2$ [6,11,14]. In case of considering only faulty vertices, Chan et al. [4] showed that a fault-free cycle of length of at least $2^n - 2f_v$ can be embedded in Q_n with $f_v \leq \lfloor (n+1)/2 \rfloor$. Yang et al. [23] further showed that a fault-free cycle of length of at least $2^n - 2f_v$ can be embedded in Q_n with $1 \leq f_v \leq n-2$. Quite recently, Fu [7] showed that a fault-free cycle of length of at least $2^n - 2f_v$ can be embedded in Q_n with $f_v \leq 2n-4$. In case of considering both faulty vertices and faulty edges, Tseng [18] showed that a fault-free cycle of length of at least $2^n - 2f_v$ can be embedded in Q_n with $f_e \leq n-4$ and $f_v + f_e \leq n-1$. Sengupta [15] generalized the above result by showing that a fault-free cycle of length $2^n - 2f_v$ can be embedded in Q_n with $f_v > 0$ or $f_e \leq n-2$, and $f_v + f_e \leq n-1$.

In this paper, we extend the result of Fu [7] to show that a fault-free cycle of length of at least $2^n - 2f_v$ can be embedded in Q_n with $f_e \leq n-2$ and $f_v + f_e \leq 2n-4$. Therefore, our result improves the previously best known result of Sengupta [15] by tolerating more faults from $n-1$ to $2n-4$.

2 Preliminaries

A *graph* $G = (V, E)$ is an ordered pair in which V is a finite set and E is a subset of $\{(u,v) |\ (u,v)$ is an unordered pair of $V\}$. We say that V is the *vertex set* and E is the *edge set*. In this paper, a network topology is represented by a simple undirected graph, which is loopless and without multiple edges. Each vertex represents a processor and each edge represents a communication link connecting a pair of processors in a network. A graph $G = (V_0 \cup V_1, E)$ is *bipartite* if $V_0 \cap V_1 = \emptyset$ and $E \subseteq \{(x,y) |\ x \in V_0$ and $y \in V_1\}$. A path $\langle v_0, v_1, v_2, \ldots, v_k \rangle$ is a sequence of vertices in which $v_0, v_1, \ldots, v_k$ are all distinct and v_i and v_{i+1} are adjacent for $0 \leq i \leq k-1$. A path $\langle v_0, v_1, v_2, \ldots, v_k \rangle$ forms a *cycle* if $v_0 = v_k$ and $v_1, v_2, \ldots, v_k$ are distinct. For graph-theoretic terminologies and notations not mentioned here, see [21].

Definition 1. An *n-dimensional hypercube* (*n-cube* for short), denoted by Q_n, is a simple undirected graph with 2^n vertices each labelled with a distinct binary string $b_1 b_2 \ldots b_{n-1} b_n$, where $b_i \in \{0,1\}$. Vertex $b_1 b_2 \ldots b_i \ldots b_{n-1} b_n$ and vertex $b_1 b_2 \ldots \overline{b_i} \ldots b_{n-1} b_n$ are connected by an edge along *dimension* i (an *i-dimensional edge* for short), where $1 \leq i \leq n$ and $\overline{b_i}$ is the one's complement of b_i.

Note that Q_n is a bipartite graph with two equal-size partite sets, and it is vertex-symmetric and edge-symmetric [12].

Let $X = x_1x_2 \dots x_i \dots x_{n-1}x_n$ and $Y = y_1y_2 \dots y_i \dots y_{n-1}y_n$ be two vertices of Q_n. In the remaining of this paper, we use $X^{(i)}$ to denote $x_1x_2 \dots \overline{x_i} \dots x_{n-1}x_n$ and use $d_H(X,Y)$ to denote the *Hamming distance* between X and Y, i.e. the number of different bits between X and Y. For convenience, an n-cube Q_n can be represented with $\underbrace{**\dots**}_{n} = *^n$, where $* \in \{0,1\}$ means the "*don't care*" symbol. Moreover, $*^{i-1}0*^{n-i}$ and $*^{i-1}1*^{n-i}$, which contain the vertices with the ith bits 0 and 1, respectively, represent two vertex-disjoint $(n-1)$-cubes.

Definition 2. An *i-partition on* $Q_n = *^n$, where $1 \le i \le n$, is to partition Q_n along dimension i into two $(n-1)$-cubes $*^{i-1}0*^{n-i}$ and $*^{i-1}1*^{n-i}$. Moreover, the edges of Q_n between $*^{i-1}0*^{n-i}$ and $*^{i-1}1*^{n-i}$ are said to be *crossing edges.*

In this paper, we consider *faulty* Q_n, i.e. Q_n contains both faulty vertices and faulty edges. A vertex (edge) is *fault-free* if it is not faulty. A path (cycle) is *fault-free* if it contains neither faulty vertex nor faulty edge. Throughout this paper, we use f_v (respectively, f_e) to denote the number of faulty vertices (respectively, edges) of the given graph. Suppose that there are $f_v \ge 1$ faulty vertices in a bipartite graph G. It is not difficult to see that the length of a longest cycle embedded into G is at most $n(G) - 2f_v$ in the worst case, where $n(G)$ is the number of vertices of G. Since the Q_n is bipartite, the length of a longest cycle embedded into Q_n with f_v faulty vertices is at most $2^n - 2f_v$, in the worst case.

3 Longest Path Embedding

In this section, we derive some properties for fault-free longest path embedding, which are useful to our cycle embedding described in the next section.

Lemma 1. [8] *Let X and Y be two arbitrary distinct fault-free vertices in an n-cube Q_n and $d_H(X,Y) = d$, where $n \ge 1$. There are X-Y paths in Q_n whose lengths are $d, d+2, d+4, ..., c$, where $c = 2^n - 1$ if d is odd, and $c = 2^n - 2$ if d is even.*

Let X and Y be two arbitrary distinct fault-free vertices of Q_n, where $n \ge 3$. A path between X and Y is abbreviated as an X-Y path. It is well known that $d_H(X,Y)$ equals the length of a shortest path between X and Y (the *distance* of X and Y) [12]. Tsai et. al. [17] showed that a fault-free X-Y path of length at least $2^n - 1$ (respectively, $2^n - 2$) can be embedded into Q_n with $f_e \le n-2$, when $d_H(X,Y)$ is odd (respectively, even). On the other hand, Fu [7] showed that a fault-free X-Y path of length at least $2^n - 2f_v - 1$ (respectively, $2^n - 2f_v - 2$) can be embedded into Q_n with $f_v \le 1$, when $d_H(X,Y) = 1$ or 3 (respectively, $d_H(X,Y) = 2$). Combing the above two results, we obtain the following lemma.

Lemma 2. *Let X and Y be two arbitrary distinct fault-free vertices of Q_n with $f_v + f_e \leq 1$, where $n \geq 3$. Then, there is a fault-free X-Y path in Q_n whose length is at least $2^n - 2f_v - 1$ (respectively, $2^n - 2f_v - 2$), when $d_H(X,Y) = 1$ or 3 (respectively, $d_H(X,Y) = 2$).*

We next show that the above lemma can be further extended. Two preliminary lemmas are first provided below.

Lemma 3. *Suppose that an n-cube Q_n contains at least two faulty vertices. Then, there exists a partition which can partition Q_n into two $(n-1)$-cubes such that each contains at least one faulty vertex.*

Proof. Straightforward. □

For an edge (U,V) (respectively, a vertex W) in a path P, the notation $P-(U,V)$ (respectively, $P-W$) means to delete (U,V) (respectively, W) from P. For paths $P_1 = \langle x_1, x_2, \ldots, x_k \rangle$ and $P_2 = \langle y_1, y_2, \ldots, y_l \rangle$, where $x_k = y_1$ and $x_1, x_2, \ldots, x_k (= y_1), y_2, \ldots, y_l$ are all distinct, the notation $P_1 + P_2$ represents the *path-concatenation*, which is an operation used to form a longer path $\langle x_1, x_2, \ldots, x_k, y_2, y_3, \ldots, y_l \rangle$.

Suppose that the given n-cube $*^n$ is partitioned into two $(n-1)$-cubes $*^{i-1}0*^{n-i}$ and $*^{i-1}1*^{n-i}$, along the ith dimension. In the remainder of this paper, we use f_v^0 (respectively, f_e^0) to denote the number of faulty vertices (respectively, faulty edges) in $*^{i-1}0*^{n-i}$. The notations f_v^1 and f_e^1 are defined similarly.

It is not difficult to see that Q_n is bipartite with two partite sets of equal size. Moreover, for two arbitrary distinct vertices X and Y, if $d_H(X,Y)$ is odd (respectively, even), then there are in different partite sets (respectively, the same partite set).

Lemma 4. *Let X and Y be two arbitrary distinct fault-free vertices of Q_n with $f_v \leq n-2$, where $n \geq 3$. Then, there is a fault-free X-Y path in Q_n whose length is at least $2^n - 2f_v - 1$ when $d_H(X,Y)$ is odd.*

Proof. The lemma can be shown by induction on n. □

The following theorem shows that a longest fault-free X-Y path can be embedded in Q_n with $f_v + f_e \leq n-2$, where $d_H(X,Y)$ is odd.

Theorem 1. *Let X and Y be two arbitrary distinct fault-free vertices in Q_n with $f_v + f_e \leq n-2$, where $n \geq 3$. Then, there is a fault-free X-Y path whose length is at least $2^n - 2f_v - 1$, when $d_H(X,Y)$ is odd.*

Proof. We show the lemma by induction on n. By Lemma 2, the basis holds. Assume that the lemma holds for $k \geq 3$. We now consider the following cases for $n = k+1$.

Case 1. $f_e = 0$. Then, $f_v \leq n-2$. By Lemma 4, the result holds.

Case 2. $f_e \neq 0$. Then, we can find a faulty i-dimensional edge and execute an i-partition to Q_{k+1} to form two k-cubes. Since the above faulty edge is a crossing edge, each k-cube contains at most $k-2$ faults. As with the proof similar to that of Lemma 4 to consider the cases of X and Y being in the same k-cube or in different k-cubes, we can show that the given Q_{k+1} contains a fault-free X-Y path of length at least $2^{k+1}-2f_v-1$ when $d_H(X,Y)$ is odd[1]. □

4 Cycle Embedding

Our method is based on a simple and efficient recursive construction. The key idea is to partition n-cube into two $(n-1)$-cubes and then construct two fault-free vertex-disjoint cycles in the two $(n-1)$-cubes. Finally, we merge the two above cycles to form a desired cycle. Our merging steps are according to different fault distributions. We next show our main results.

Theorem 2. *Let $n \geq 3$ be an integer. There exists a fault-free cycle of length at least $2^n - 2f_v$ in Q_n with $1 \leq f_e \leq n-2$ and $1 \leq f_v + f_e \leq 2n-4$.*

Proof. The proof is by induction on n. It is not difficult to verify by a computer program that Q_3 contains a fault-free cycle of length at least $8-2f_v$ when $1 \leq f_v + f_e \leq 2$ and $f_e = 1$. Thus the basis case is true. Assume that the result is true for $n = k$. We now consider that $n = k+1$.

Since $f_e \geq 1$, we can select an arbitrary faulty edge. Assume that such a faulty edge is of dimension i. By executing an i-partition on Q_{k+1}, we obtain two k-cubes $*^{(i-1)}0*^{(k-i+1)}$ and $*^{(i-1)}1*^{(k-i+1)}$. Since at least one faulty edge belongs to crossing edges, we have that $f_v^i + f_e^i \leq 2k-3$ and $f_e^i \leq k-2$ for $i = 0, 1$. We next consider the following cases.

Case 1. $f_v^0 + f_e^0 = 0$ and $f_v^1 + f_e^1 = 2k-3$. In this case, one faulty edge is a crossing edge. Since $f_e^1 \leq k-2$ and $f_v^1 + f_e^1 = 2k-3$, we have that $|f_v^1| > 0$. We regard one faulty vertex in $*^{(i-1)}1*^{(k-i+1)}$ as fault-free. Then, by induction hypothesis, there is a cycle C in $*^{(i-1)}1*^{(k-i+1)}$, whose length is at least $2^k - 2f_v'$, where $f_v' = f_v^1 - 1$. Note that C contains at most one faulty vertex. Without loss of generality, assume that C contains one faulty vertex F. Let U and V be two vertices adjacent to F in C.

Case 1.1. Both $(U, U^{(i)})$ and $(V, V^{(i)})$ are fault-free. Note that $d_H(U^{(i)}, V^{(i)}) = 2$. By Lemma 2, there is a fault-free $U^{(i)}$-$V^{(i)}$ path P in $*^{(i-1)}\,0\,*^{(k-i+1)}$, whose length is at least $2^k - 2f_v^0 - 2 = 2^k - 2$. Then, a fault-free cycle of Q_{k+1} can be constructed as $(C-F) + (U, U^{(i)}) + P + (V, V^{(i)})$, whose length is at least $(2^k-2)+2+2^k-2(f_v^1-1)-2 = 2^{k+1}-2(f_v^1) = 2^{k+1}-2f_v$.

Case 1.2. Either $(U, U^{(i)})$ or $(V, V^{(i)})$ is faulty. The proof is similar to the above case.

[1] Since the proof is very similar to that of Lemma 4, we omit here.

Case 2. $1 \leq f_v^0 + f_e^0 \leq k-2$. In this case, we have that $f_v^1 + f_e^1 \leq 2k-4$. By induction hypothesis, the k-cube $*^{(i-1)}1*^{(k-i+1)}$ contains a fault-free cycle C of length at least $2^k - 2f_v^1$. Since $f_v + f_e \leq 2k-2$, it is not difficult to show that there is an edge (U,V) in C such that $U^{(i)}$ and $V^{(i)}$ are both fault-free. (If no such an edge exists, then $f_v^0 + f_e^0 \geq \lfloor \frac{2^k - 2f_v^1}{2} \rfloor = 2^{k-1} - f_v^1$. Thus $f_v + f_e \geq f_v^0 + f_e^0 + f_v^1 \geq 2^{k-1} > 2k-2$ for $k \geq 4$, which contradicts to the assumption that $f_v + f_e \leq 2n-4 = 2k-2$). Moreover, since $f_v^0 + f_e^0 \leq k-2$, there is a fault-free $U^{(i)}$-$V^{(i)}$ path P of length at least $2^k - 2f_v^0 - 1$ by Theorem 1. Therefore, a fault-free cycle of Q_{k+1} can be constructed as $(C-(U,V)) + (U,U^{(i)}) + P + (V,V^{(i)})$ whose length is at least $(2^k - 2f_v^1 - 1) + 2 + (2^k - 2f_v^0 - 1) = 2^{k+1} - 2(f_v^0 + f_v^1) = 2^{k+1} - 2f_v$.

Case 3. $f_v^0 + f_e^0 = 0$ and $f_v^1 + f_e^1 \leq 2k-4$. In this case, the crossing edges between $*^{(i-1)}0*^{(k-i+1)}$ and $*^{(i-1)}1*^{(k-i+1)}$ contain more than one faulty edges. A desired fault-free cycle can be constructed with the method similar to that of Case 2.

Case 4. $k-1 \leq f_v^0 + f_e^0 \leq 2k-4$. Since $f_v^0 + f_e^0 \geq k-1$ and there is at least one faulty edge belong to the crossing edges, we have that $f_v^1 + f_e^1 \leq k-2$. Therefore, it is not difficult to construct a desired fault-free cycle with the method similar to that described in Case 2.

Case 5. $f_v^0 + f_e^0 = 2k-3$ and $f_v^1 + f_e^1 = 0$. This case is symmetric to Case 1.

Combining the above cases, we complete the proof. □

Lemma 5. [7] *Let $n \geq 3$ be an integer. There exists a fault-free cycle of length of at least $2^n - 2f_v$ in Q_n with $f_v \leq 2n-4$.*

By combining Theorem 2 and Lemma 5, we have the following theorem.

Theorem 3. *Let $n \geq 3$ be an integer. There exists a fault-free cycle of length at least $2^n - 2f_v$ in Q_n with $f_e \leq n-2$ and $f_e + f_v \leq 2n-4$.*

Proof. If $f_e = 0$, then the result follows from Lemma 5. Otherwise, if $f_e \neq 0$, then the result follows from Theorem 2. □

References

1. S. G. Akl, *Parallel Computation: Models and Methods*. Prentice Hall, 1997.
2. B. Alspach, J. C. Bermond, and D. Sotteau, "Decomposition into cycles I. Hamiltonian decomposition," Technical Report 87-12, Simon Fraser University (1987).
3. J. Bruck, R. Cypher, and D. Soroker, "Embedding cube-connected-cycles graphs into faulty hypercubes," *IEEE Transactions on Computers*, vol. 43, no. 10, pp. 1210–1220, 1994.
4. M. Y. Chan and S. J. Lee, "Distributed fault-tolerant embeddings of rings in hypercubes," *Journal of Parallel and Distributed Computing*, vol. 11, pp. 63–71, 1991.
5. M. Y. Chan and S. J. Lee, "Fault-tolerant embeddings of complete binary trees in hypercubes," *IEEE Transactions on Parallel and Distributed Systems*, vol. 4, no. 3, pp. 540–547, 1993.

6. M. Y. Chen and K. G. Shin, "Processor allocation in an N-cube multiprocessor using gray codes," *IEEE Transactions on Computers*, vol. C-36, pp. 1396–1407, 1987.
7. Jung-Sheng Fu, "Fault-tolerant cycle embedding in the hypercube," *Parallel Computing*, vol. 29, pp. 821–832, 2003.
8. Jung-Sheng Fu and Gen-Huey Chen, "Hamiltonicity of the Hierarchical Cubic Network," *Theory of Computing Systems*, vol. 35, pp. 59–79, 2002.
9. S. Y. Hsieh, G. H. Chen, and C. W. Ho, "Embed longest rings onto star graphs with vertex faults," *Proceedings of the International Conference on Parallel Processing*, 1998, pp. 140–147.
10. Chien-Hung Huang, Ju-Yuan Hsiao, and R. C. T. Lee, "An optimal embedding of cycles into incomplete hypercubes," *Information Processing Letters*, vol. 72, pp. 213–218, 1999.
11. S. Latifi, S. Q. Zheng, N. Bagherzadeh, "Optimal ring embedding in hypercubes with faulty links," in *Processings of the IEEE Symposium on Fault-Tolerant Computing*, pp. 178–184, 1992.
12. F. T. Leighton, *Introduction to Parallel Algorithms and Architecture: Arrays· Trees· Hypercubes*. Morgan Kaufmann, San Mateo, CA, 1992.
13. R. A. Rowley and B. Bose, "Fault-tolerant ring embedding in deBruijn networks," *IEEE Transacions on Computers*, vol. 42, no. 12, pp. 1480–1486, 1993.
14. A. Sen, A. Sengupta, and S. Bandyopadhyay, "On some topological properties of hypercube, incomplete hypercube and supercube," in *Proceedings of the International Parallel Processing Symposium*, Newport Beach, April, pp. 636–642, 1993.
15. Abhijit Sengupta, "On ring embedding in hypercubes with faulty nodes and links," *Information Processing Letters*, vol. 68, pp. 207–214, 1998.
16. G. Tel, *Topics in distributed algorithms*. Cambridge Int'l Series on Parallel Computation, Cambridge Universitry Press, 1991.
17. Chang-Hsiung Tsai, Jimmy J.M. Tan, Tyne Liang, and Lih-Hsing Hsu, "Fault-tolerant hamiltonian laceability of hypercubes," *Information Processing Letters*, vol. 83, no. 6, pp. 301–306, 2002.
18. Yu-Chee Tseng, "Embedding a ring in a hypercube with both faulty links and faulty nodes," *Information Processing Letters*, vol. 59, pp. 217–222, 1996.
19. Yu-Chee Tseng, S. H. Chang, and J. P. Sheu, "Fault-tolerant ring embedding in star graphs with both link and node failures," *IEEE Transactions on Parallel and Distributed Systems*, vol. 8, no. 12, pp. 1185–1195, 1997.
20. Yu-Chee Tseng and T. H. Lai, "On the embedding of a class of regular graphs in a faulty hypercube," *Journal of Parellel Distributed Computing*, vol. 37, pp. 200–206, 1996.
21. D. B. West, *Introduction to Graph Theory*. Prentice-Hall, Upper Saddle River, NJ 07458, 2001.
22. P. J. Yang and C. S. Raghavendra, "Embedding and reconfiguration of binary trees in faulty hypercubes," *IEEE Transactions on Parallel and Distributed Systems*, vol. 7, no. 3, pp. 237–245, 1996.
23. P. J. Yang, S. B. Tien, and C. S. Raghavendra, "Embedding of rings and meshes onto faulty hypercubes using free dimensions," *IEEE Transactions on Computers*, vol. 43, no. 5, pp. 608–613, 1994.

Improving the Performance of GCC by Exploiting IA-64 Architectural Features*

Canqun Yang[1], Xuejun Yang[1], and Jingling Xue[2]

[1] School of Computer Science, National University of Defense Technology, Changsha, Hunan 410073, China
[2] Programming Languages and Compilers Group, School of Computer Science and Engineering, The University of New South Wales, Sydney, NSW 2052, Australia

Abstract. The IA-64 architecture provides a rich set of features to aid the compiler in exploiting instruction-level parallelism to achieve high performance. Currently, GCC is a widely used open-source compiler for IA-64, but its performance, especially its floating-point performance, is poor compared to that of commercial compilers because it has not fully utilized IA-64 architectural features. Since late 2003 we have been working on improving the performance of GCC on IA-64. This paper reports four improvements on enhancing its floating-point performance, namely alias analysis for FORTRAN (its part for COMMON variables already committed in GCC 4.0.0), general induction variable optimization, loop unrolling and prefetching arrays in loops. These improvements have significantly improved the floating-point performance of GCC on IA-64 as extensively validated using SPECfp2000 and NAS benchmarks.

1 Introduction

Based on EPIC (Explicitly Parallel Instruction Computing) technology, the IA-64 architecture[8,9] was designed to allow the compiler explicit control over the execution resources of the processor in order to maximize instruction-level parallelism (ILP). To achieve this, the IA-64 architecture provides a rich set of architectural features to facilitate and maximize the ability of the compiler to expose, enhance and exploit instruction-level parallelism (ILP). These features include speculation, predication, register rotation, advanced branch architecture, special instructions such as data prefetching, post-increment loads and stores, and many others. Thus, the compiler's ability to deliver many of the functionalities that are commonly realized in the processor hardware greatly impacts the performance of the processors in the IA-64 family.

GCC (GNU Compiler Collection) is an open-source, multi-language and multi-platform compiler. Being fairly portable and highly optimizing, GCC is

* This work was supported by the National High Technology 863 Program of China under Grant 2004AA1Z2210.

T. Srikanthan et al. (Eds.): ACSAC 2005, LNCS 3740, pp. 236–251, 2005.

widely used in research, business, industry and education. However, its performance, especially its floating-point performance, on the IA-64 architecture, is poor compared to that of commercial compilers such as Intel's icc [4]. We have measured the performance results of GCC (version 3.5-tree-ssa) and icc (version 8.0) using SPEC CPU2000 benchmarks on a 1.0 GHz Itanium 2 system. GCC has attained 70% of the performance of icc for SPECint2000. In the case of SPECfp2000, however, the performance of GCC has dropped to 30% of that of icc. Since 2001 several projects have been underway on improving the performance of GCC on IA-64 [16]. While the overall architecture of GCC has undergone some major changes, its performance on IA-64 has not improved much.

This paper describes some progress we have made in our ongoing project on improving the performance of GCC on the IA-64 architecture. Commercial compilers such as Intel's icc and HP's compilers are proprietary. Research compilers such as ORC [15] and openIMPACT [14] are open-source but include only a few frontends (some of which are not extensively tested). GCC is attractive to us since it is an open-source, portable, multi-language and multi-platform compiler. We are interested in IA-64 partly because it is a challenging platform for compiler research and partly because of our desire in developing also an open-source compiler framework for VLIW embedded processors.

In late 2003, we initiated this project on improving the performance of GCC on IA-64. We have done most of our research in GCC 3.5-tree-ssa. As this version fails to compile many SPECfp2000 and NAS benchmarks, we have fixed the FORTRAN frontend so that all except the two SPECfp2000 benchmarks, `fma3d` and `sixtrack`, can compile successfully. We are currently porting our work to GCC 4.0.0.

In this paper, we report four improvements we have incorporated into GCC for improving its floating-point performance, namely, alias analysis for FORTRAN, general induction variable optimization, loop unrolling and prefetching arrays in loops. Our alias analysis for COMMON variables has already been committed in GCC 4.0.0. The four improvements were originally implemented in GCC 3.5 and have recently been ported to GCC 4.0.0 as well. In GCC 3.5, we have observed a performance increase of 41.8% for SPECfp2000 and 56.1% for the NAS benchmark suite on a 1.0 GHz Itanium 2 system. In GCC 4.0.0, its new loop unrolling has a performance bug: it does not (although it should have) split induction variables as it did in GCC 3.5. This affects the benefit of our loop unrolling negatively in some benchmarks. Our improvements incorporated into GCC 4.0.0 have resulted a performance increase of 14.7% for SPECfp2000 and 32.0% for NAS benchmark suite, respectively. Finally, GCC 3.5 (with our four improvements included) outperforms GCC 4.0.0 (the latest GCC release) by 32.5% for SPECfp2000 and 48.9% for NAS benchmark suite, respectively.

The plan of this paper is as follows. Section 2 reviews the overall structure of GCC. Section 3 discusses its limitations that we have identified and addressed in this work. In Section 4, we present our improvements for addressing these limitations. In Section 5, we present the performance benefits of all our improvements

for SPECfp2000 and NAS benchmarks on an Itanium 2 system. Section 6 reviews the related work. Section 7 concludes the paper and discusses some future research directions.

2 GCC Overview

GCC consists of language-specific frontends, a language-independent backend and architecture-specific machine descriptions [18,20]. The frontend for a language translates a program in that language into an abstract syntax tree called *GIMPLE*. High-level optimizations, such as alias analysis, function inlining, loop transformations and partial redundancy elimination (PRE), are carried out on GIMPLE. However, high-level optimizations in GCC are limited and are thus topics of some recent projects [5].

Once all the tree-level optimizations have been performed, the syntax tree is converted into an intermediate representation called *RTL* (Register Transfer Language). Many classic optimizations are done at the RTL level, including strength reduction, induction variable optimization, loop unrolling, prefetching arrays in loops, instruction scheduling, register allocation and machine-dependent optimizations. In comparison with the tree-level optimizations (done on GIMPLE), the RTL-to-RTL passes are more comprehensive and more effective in boosting application performance.

Finally, the RTL representation is translated into assembly code. A *machine description* for a target machine contains all machine-specific information consulted by various compiler passes. Such a description consists of instruction patterns used for generating RTL instructions from GIMPLE and for generating assembly code after all RTL-to-RTL passes. Properly defined instruction patterns can help generate optimized code. In addition, a machine description also contains macro definitions for the target processor (e.g., processor architecture and function calling conventions).

3 Limitations of GCC on IA-64

The performance of GCC is quite far behind that of icc: GCC 3.5 reaches only 70% and 30% of icc 8.0 in SPECint2000 and SPECfp2000, respectively. Compared to icc, GCC 3.5 lacks loop transformations such as loop interchange, loop distribution, loop fusion and loop tiling, software pipelining and interprocedural optimizations. These three kinds of important optimizations are critical for icc's performance advantages. The importance of these optimizations for improving the performance of GCC was noted [16]. However, little progress has been made in GCC 4.0.0. The function inlining remains the only interprocedural optimization supported in GCC 4.0.0. The SWING modulo scheduler [7] was included in GCC 4.0.0 to support software pipelining. According to our experimental results, it cannot successfully schedule any loops in the SPECfp2000 and NAS benchmarks on IA-64. Loop interchange was added in GCC 4.0.0 but again it has not interchanged any loops in the SPECfp2000 and NAS benchmarks on IA-64.

One of our long-term goals is to develop and implement these three kinds of important optimizations in GCC to boost its performance on IA-64. At the same time, we will try to maintain the competitive edge of GCC as an open-source, multi-language and multi-platform compiler. In this early stage of our project, our strategy is to identify some limitations in the current GCC framework so that their refinements can lead to significant increase in the floating-point performance of GCC on IA-64. We have analyzed extensively the performance results of benchmarks compiled under different optimization levels and different (user-invisible) tunable optimization parameters in GCC using tools such as `gprof` and `pfmon`. We have also analyzed numerous assembly programs generated by GCC. The following two problem areas of GCC on IA-64 are identified:

- There is no alias analysis for FORTRAN programs in GCC. The lack of alias information reduces opportunities for many later RTL-level optimizations.
- The loop optimizations in GCC are weak. In particular, general induction variable optimization, loop unrolling and prefetching arrays in loops do not fully utilize IA-64 architectural features in exposing and exploiting instruction-level parallelism. Due to the lack of sophisticated high-level optimizations, the effectiveness of these RTL optimizations can be critical to the overall performance of GCC.

4 Improvements of GCC for IA-64

In this section, we present our improvements for the four components of GCC that we identified in Section 3 in order to boost its floating-point performance significantly. These four components are alias analysis for FORTRAN, general induction variable optimization, loop unrolling and prefetching arrays in loops. The alias analysis is performed on GIMPLE while the three optimizations are done at the RTL level.

We describe our improvements to the four components of GCC in separate subsections. In each case, we first describe the current status of GCC, then present our solution, and finally, evaluate its effectiveness using some selected benchmarks. Once having presented all the four improvements, we discuss the performance results of our improvements for the SPECfp2000 benchmark suite and the NAS benchmark suite. In this section, all benchmarks are compiled under GCC 3.5 at "-O3" on an Itanium 2 system, whose hardware details can be found in Section 5.

4.1 Alias Analysis

Alias analysis refers to the determination of storage locations that may be accessed in more than one way. Alias information is generally gathered by the front-end of the compiler and passed to the back-end to guide later compile optimizations. In GCC, alias analysis has been implemented for C/C++ but not for FORTRAN. This section introduces a simple alias analysis module we have added for FORTRAN in GCC.

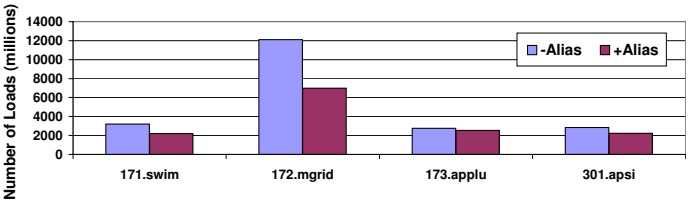


Fig. 1. Load instructions retired with (+Alias) and without (-Alias) alias analysis (in GCC 3.5)

GCC conducts alias analysis at the tree level (i.e., on GIMPLE) via an interface function, LANG_HOOKS_GET_ALIAS_SET, common to all programming languages. Each language-specific frontend provides its own implementation of this interface function. We have completed a simple implementation of an intraprocedural alias analysis for FORTRAN, by mainly detecting the aliases created due to EQUIVALENCE statements, pointers, objects with TARGET attributes and parameters.

In GCC, an alias set contains all memory references that are aliases to each other. Two memory references in different alias sets are not aliases. In our intraprocedural alias analysis, the alias sets are constructed based on the following simple facts:

- A COMMON variable that is contained in a COMMON block is its own alias set if there are not EQUIVALENCE objects within this COMMON block.
- There are no aliases for a parameter of a function (except the parameter itself) if the compiler switch "-fargument-noalias" is enabled by the user.
- A local variable is in its own alias set if it is not a pointer and does not have a TARGET attribute.

Figure 1 shows that such a simple alias analysis is already effective in removing redundant load instructions. These results for the four SPECfp2000 benchmarks compiled under GCC 3.5 at "-O3" are obtained using `pfmon` running with the train inputs. The percentage reductions for the four benchmarks `swim`, `mgrid`, `applu` and `apsi` are 31.25%, 42.15%, 8.00% and 21.13%, respectively.

4.2 General Induction Variable Optimizations

Induction variables are variables whose successive values form an arithmetic progression over some part (usually a loop) of a program. They are often divided into two categories: basic induction variables (BIVs) and general induction variables (GIVs). A BIV is modified (incremented or decremented) explicitly by the same constant amount during each iteration of a loop. A GIV may be modified in a more complex manner. There are two kinds of optimizations for induction

variables: induction variable elimination and strength reduction. We improve the efficiency of the strength reduction of GIVs on IA-64 by utilizing some IA-64 architectural features.

GCC can identify the *GIV* of the form $b + c \times I$. If this is the address of an array, then b represents the base address of the array, I a loop variable and c the size of the array element. An address GIV can be strength reduced by replacing the multiplication $c \times I$ in the GIV with additions. This enables the array access operation and the address increment/decrement to be combined into one instruction. Such an optimization has been implemented in GCC. However, there are no great performance improvements on IA-64 since the legality test required for the optimization is too conservative.

In programs running on IA-64, the loop variable I (a BIV) is typically a 32-bit integer variable while the address of an array element (a GIV) is typically 64-bit long. The address $b+c\times I$ is normally computed as follows. First, the BIV I is evaluated and extended into 64 bits. Then $b + c \times I$ is evaluated to obtain the address GIV. Before performing the strength reduction for the address GIV, GCC first checks to see if the BIV may overflow or not (as a 32-bit integer) during loop execution. If the BIV may overflow, then whether the GIV can be legally reduced or not depends on whether I is unsigned or signed. In programming languages such as C, C++ and FORTRAN, unsigned types have the special property of never overflowing in arithmetic. Therefore, the strength reduction for the address GIV as discussed above may not be legal. For signed types, the semantics for an overflow in programming languages are usually undefined. In this case, GCC uses a compiler switch to determine if the strength reduction can be performed or not. If "-fwrapv" is turned on, then signed arithmetic overflow is well-defined. The strength reduction for the address GIV may not be legal if the BIV may overflow. If "-fwrapv" is turned off, then the strength reduction can be performed. In GCC, the function that performs the legality test for the GIV strength reduction is check_ext_dependent_givs. When compiling FORTRAN programs, the outcome of such a legality test is almost always negative. This is because the FORTRAN frontend introduces a temporary to replace the BIV I, causing the test to fail in general.

We have made two refinements for this optimization. First, the strength reduction for an address GIV is always performed if "-fwrapv" is turned off (which is the default case). The BIVs are signed in FORTRAN programs. This refinement yield good performance benefits for some benchmarks. Second, for unsigned BIVs (as in C benchmarks), we perform a limited form of symbolic analysis to check if these BIVs may overflow or not. If they do not overflow, then the GIV strength reduction can be performed.

Figure 2 illustrates the performance impact of the improved GIV optimization on four SPECfp2000 benchmarks. These benchmarks are compiled under GCC 3.5 at "-O3" with our alias analysis being enabled. The cycle distributions on the Itanium 2 processors are obtained as per [19]. Reducing the strength for an address GIV creates the opportunity for the array address access and address increment/decrement operations to be merged into one instruction. Therefore,

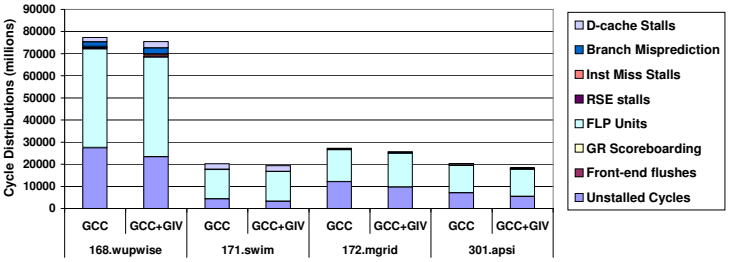


Fig. 2. Effects of the improved GIV optimization on Itanium cycle categories (in GCC 3.5)

unstalled cycles for the four benchmarks are significantly reduced. The percentage reductions for `wupwise`, `swim`, `mgrid` and `apsi` are 14.82%, 25.34%, 19.59% and 23.96%, respectively.

4.3 Loop Unrolling

Loop unrolling for a loop replicates the instructions in its loop body into multiple copies. In addition to reduce the loop overhead, loop unrolling can also improve the effectiveness of other optimizations such as common subexpression elimination, induction variable optimizations, instruction scheduling and software pipelining. Loop unrolling is particularly effective in exposing and enhancing instruction-level parallelism.

In GCC, the effectiveness of loop unrolling is crucially dependent on a tunable parameter called MAX_UNROLLED_INSNS. This parameter specifies the maximum number of (RTL) instructions that is allowed in an unrolled loop. The default is 200.

The existing loop unrolling algorithm in GCC works as follows. Let LOOP_CNT be the number of iterations in a loop. Let NUM_INSNS be the number of instructions in a loop. Let UNROLL_FACTOR be the number of times that the loop is unrolled. UNROLL_FACTOR is chosen so that the following condition always holds:

$$\text{NUM\_INSNS} \times \text{UNROLL\_FACTOR} < \text{MAX\_UNROLLED\_INSNS} \quad (1)$$

The situation when the exact value of LOOP_CNT can be calculated statically (i.e., at compile time) is handled specially. The loop will be fully unrolled when

$$\text{NUM\_INSNS} \times \text{LOOP\_CNT} < \text{MAX\_UNROLLED\_INSNS} \quad (2)$$

Otherwise, UNROLL_FACTOR is set as the largest divisible factor of LOOP_CNT such that (1) holds. If UNROLL_FACTOR has not been determined so far or LOOP_CNT can only be calculated exactly at run time, UNROLL_FACTOR is set as the largest in $\{2, 4, 8\}$ such that (1) holds. Finally, the

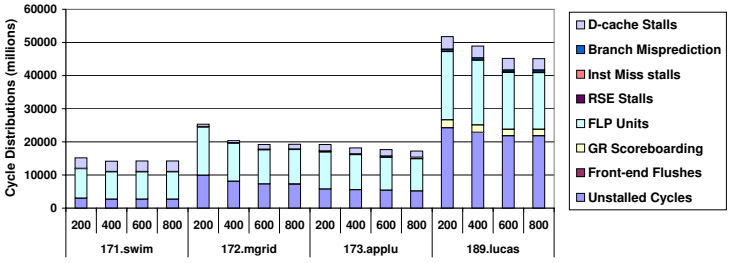


Fig. 3. Effects of the improved loop unrolling on Itanium cycle categories (in GCC 3.5)

so-called preconditioning code is generated for a loop whenever possible so that only one exit test is needed in the unrolled loop.

Loop unrolling on IA-64 is not effective since the default 200 for MAX_UNROLLED_INSNS is inappropriate for this architecture. We have done extensive benchmarking by trying different values. We found that loop unrolling is the most effective on IA-64 if MAX_UNROLLED_INSNS is set to be 600. Figure 3 gives the cycle distributions of four SPECfp2000 benchmarks compiled under GCC 3.5 at "-O3 -funroll-loops" with improved alias analysis and GIV optimization and run under the train inputs when MAX_UNROLLED_INSNS takes four different values.

As shown in the experimental results, loop unrolling becomes more effective when performed more aggressively on IA-64. Unrolling more iterations in a loop tends to increase the amount of instruction-level parallelism in the loop. As a result, the number of unstalled cycles and the number of cycles spent on the FLP units are both reduced. The best performance results for the four benchmarks are attained when MAX_UNROLLED_INSNS = 600. However, loop unrolling may increase register pressure and code size. As MAX_UNROLLED_INSNS increases, more loops and larger loops may be unrolled, leading to potentially higher register pressure and larger code size. Fortunately, the IA-64 architecture possesses large register files and can sustain higher register pressure than other architectures. So we recommend MAX_UNROLLED_INSNS to be set as 600. In future work, we will investigate a more sophisticated strategy that can also take register pressure into account.

4.4 Prefetching Arrays in Loops

Data prefetching techniques anticipates cache misses and issue fetches to the memory system in advance of the actual memory accesses. To provide a overlap between processing and memory accesses, computation continues while the prefetched data are being brought into the cache. Data prefetching is therefore complementary to data locality optimizations such as loop tiling and scalar replacement.

In GCC, the array elements in loops are prefetched at the RTL level. However, its prefetching algorithm is not effective on IA-64. The prefetching algorithm relies on a number of tunable parameters. Those relevant to this work are summarised below.

1. PREFETCH_BLOCK specifies the cache block size in bytes for the cache at a particular level. The default value is 32 bytes for the caches at all levels.
2. SIMULTANEOUS_PREFETCHES specifies the maximum number of prefetch instructions that can be inserted into an innermost loop. If more prefetch instructions are needed in an innermost loop, then no prefetch instructions will be issued at all for the loop. The default value on IA-64 is 6, which is equal to the maximum number of instructions that can be issued simultaneously on IA-64.
3. PREFETCH_BLOCKS_BEFORE_LOOP_MAX specifies the maximum of prefetch instructions inserted before a loop (to prefetch the cache blocks for the first few iterations of the loop). The default value is also 6.
4. PREFETCH_DENSE_MEM represents the so-called memory access density for a prefetch instruction. It refers to the ratio of the number of bytes actually accessed to the number of bytes prefetched in a prefetch instruction. In Itanium 2, the cache line sizes of its L1, L2 and L3 caches are 64 bytes, 128 bytes and 128 bytes, respectively. It is therefore possible that some data prefetched by a prefetch instruction may not be accessed. Thus, PREFETCH_DENSE_MEM reflects the effectiveness of a single prefetch instruction. The default value for this parameter is 220/256.

Our experimental evaluations show that the default values for the first three parameters are not reasonable. Figure 4 plots some statistics about prefetch instructions required inside loops in four SPECfp2000 benchmarks. A data point $(x, y\%)$ for a benchmark means that the percentage number of loops requiring x or fewer prefetch instructions in that benchmark is $y\%$. The statistics are obtained using GCC 3.5 at the optimization level "-O3 -funroll-loops -fprefetch-loop-arrays" with the alias analysis for FORTRAN, GIV optimization and loop unrolling incorporated. In `swim`, the number of prefetch instructions required by 81.00% of the loops is less than or equal to 6, but these loops account for a small

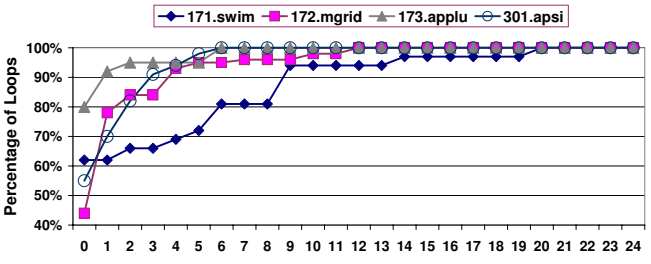


Fig. 4. Prefetch instructions required in four SPECfp2000 benchmarks (in GCC 3.5)

portion of the execution time of the program. The loops that are responsible for the most of the execution time may require 7 or more prefetch instructions. For example, loop 100 in function CALC1, loop 200 in function CALC2 loop 300 in function CACL3 and loop 400 in function CACL3Z require 14, 20, 9 and 9 prefetch instructions, respectively. In `mgrid`, loop 600 in function PSINV and loop 800 in function RESID account for nearly all the execution time. They require 10 and 12 prefetch instructions, respectively. In all these loops requiring more than 6 prefetch instructions, no instructions will be actually inserted according to the semantics of SIMULTANEOUS_PREFETCHES.

Therefore, it is too simplistic to use a uniform upper bound to limit the number of prefetch instructions issued in all loops. Some loops may need more prefetch instructions than others. Inserting too many prefetch instructions into a loop may result in performance degradation. However, such a situation can be alleviated by adopting rotating register allocation [3]. Unfortunately, such a scheme is not supported in GCC.

We have refined the prefetching algorithm in GCC for IA-64 as follows: All the default values are chosen by extensive benchmarking on an Itanium 2 system.

- First, we should allow more prefetch instructions to be issued on IA-64:

  PREFETCH_BLOCKS_BEFORE_LOOP_MAX = 12
  SIMULTANEOUS_PREFETCHES = 12

- Second, we introduce a new parameter, PMAX, which is used to determine the maximum number of prefetch instructions, that can be inserted inside a loop:

  PMAX = MIN (SIMULTANEOUS_PREFETCHES, NUM_INSNS ÷ 6).

  where NUM_INSNS is the number of instructions in the loop. If the number of prefetch instructions calculated are no large than PMAX, then all will be issued. Otherwise, the PMAX most effective prefetch instructions will be issued. Prefetch instruction F_1 is more effective than prefetch instruction F_2 if more array accesses can use the data prefetched by F_1 than that by F_2. That being equal, F_1 is more effective than F_2 if F_1 has a higher memory access density than F_2.
- Third, there are three levels of cache in the Itanium 2 processors. The L1 cache is only for integer values. The cache line sizes for L1, L2 and L3 caches are 64, 128 and 128 bytes, respectively. We set PREFETCH_BLOCK=64 for integer values and PREFETCH_BLOCK=128 for floating-point values. In addition, we use the prefetch instruction `lfetch` to cache integer values at the L1 cache and `lfetch.nt1` to cache floating-point values at the L2 cache. Our experimental results show that this third refinement leads to only slight performance improvements in a few benchmarks. Note that the statistical results shown in Figure 4 remain nearly the same when the two different values for PREFETCH_BLOCK are used. This is because in GCC, the prefetch instructions required for an array access inside a loop is calculated as (STRIDE + PREFETCH_BLOCK - 1)/PREFETCH_BLOCK, where STRIDE is 8 bytes for almost all array accesses.

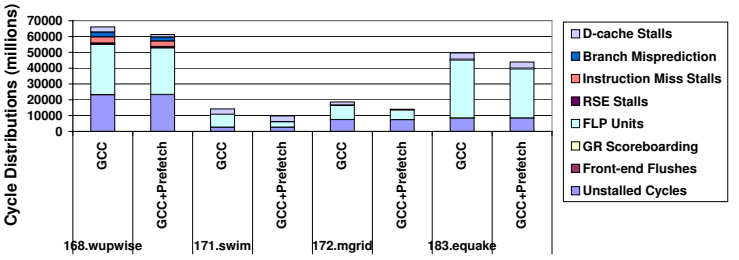


Fig. 5. Effects of the improved prefetching on Itanium cycle categories(in GCC 3.5)

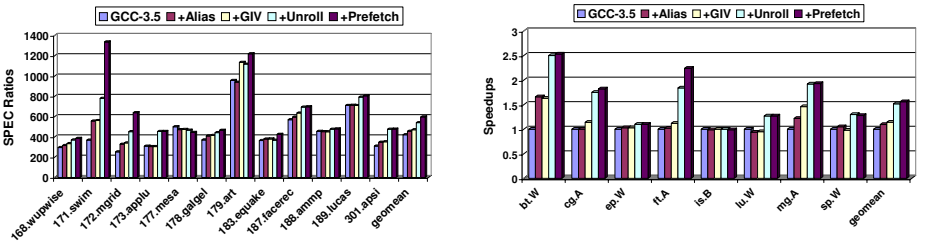
Figure 5 illustrates the effectiveness of our improved prefetching algorithm. All four benchmarks are compiled under GCC at "-O3 -funroll-loops -fprefetch-loop-arrays" with alias analysis, GIV optimization and loop unrolling included. The percentage reductions in the D-cache (L1 cache) category for `wupwise`, `swim`, `mgrid` and `equake` are 51.66%, -5.356%, 84.59% and 0.99%, respectively. The percentage reductions in the FLP units for the same four benchmarks are 7.28%, 57.38%, 31.77% and 15.45%, respectively. This category includes the stalls caused by the register-register dependences and the stalls when instructions are waiting for the source operands from the memory subsystem. By prefetching array data more aggressively in computation-intensive loops, the memory stalls in these benchmarks have been reduced more significantly.

5 Experimental Results

We have implemented our techniques in GCC 3.5 and GCC 4.0.0. We evaluate this work using SPECfp2000 and NAS benchmarks on a 1.0 GHz Itanium 2 system running Redhat Linux AS 2.1 with 2GB RAM. The system has a 16KB L1 instruction cache, a 16KB data cache, a 256KB L2 (unified) cache and a 3MB L3 (unified) cache. We have excluded the two SPECfp2000 benchmarks, `fma3d` and `sixtrack`, in our experiments since they cannot compile and run successfully under GCC 3.5. We present and discuss our results under GCC 3.5 and GCC 4.0.0 in two separate subsections.

5.1 GCC 3.5

Figure 6(a) illustrates the cumulative effects of our four techniques on improving the performance of SPECfp2000. "GCC-3.5" refers to the configuration under which all benchmarks are compiled using GCC 3.5 at "-O3 -funroll-loops -fprefetch-loop-arrays". "+Alias" stands for GCC 3.5 with the alias analysis for FORTRAN being included. In "+GIV", the GIV optimization is also enabled. In "+Unroll", our loop unrolling is also turned on. Finally, "+Prefetch" means that the optimization for prefetching arrays in loops is also turned on. Therefore, "+Prefetch" refers to GCC 3.5 with all our four techniques being enabled.



(a) SPECfp2000 (b) NAS

Fig. 6. Performance results of SPECfp and NAS benchmarks

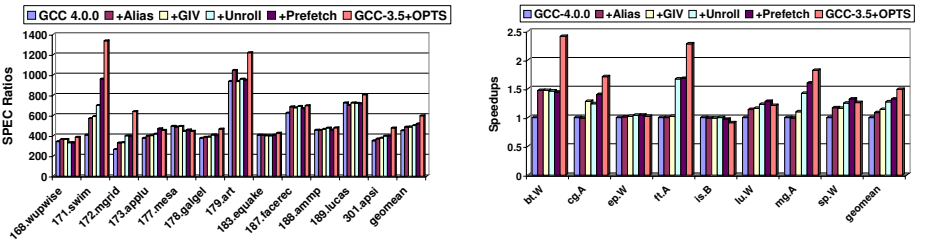
The performance of each benchmark is the (normalized) ratio of the run time of the benchmark to a SPEC-determined reference time. The ratio for SPECfp2000 is calculated as the geometric mean of the normalized ratios for all the benchmarks.

Before our optimizations are used, the ratio of SPECfp2000 is 420.7. Alias analysis helps lift the ratio to 455.1, resulting in a 8.2% performance increase for SPECfp2000. The GIV optimization pushes the ratio further to 470.1, which represents a net performance increase of 3.3%. Loop unrolling is the most effective. Once this optimization is turned on, the ratio of SPECfp2000 reaches 540.1. This optimization alone improves the SPECfp2000 performance by 14.9%. Finally, by prefetching arrays in loops, the ratio of SPECfp2000 climaxes to 596.4. This optimization is also effective since a net performance increase of 10.4% is observed. Our four optimizations have increased the ratio of SPECfp2000 from 420.7 to 596.4, resulting a performance increase of 41.8%. For SPECfp2000, loop unrolling and prefetching are the two most effective optimizations.

Figure 6(b) shows the performance improvements for the NAS benchmarks. The execution times of a benchmark under all configurations are normalized to that obtained under "GCC-3.5" (with our techniques turned off). Therefore, the Y-axis represents the performance speedups of our optimization configurations over "GCC-3.5". The performance increase for the entire benchmark suite under each configuration is taken as the geometric mean of the speedups of all the benchmarks under that configuration. The speedups for "+Alias", "+GIV", "+Unroll" and "+Prefetch" over "GCC-3.5" are 9.6%, 14.3%, 51.7% and 56.1%. Therefore, our four optimizations have resulted a 56.1% performance increase for the NAS benchmark suite. For these benchmarks, alias analysis and loop unrolling are the two most effective optimizations.

5.2 GCC 4.0.0

GCC 4.0.0 is the latest release of GCC, which includes (among others) the SWING modulo scheduler for software pipelining [7] and some loop transformations such as loop interchange. As we mentioned earlier, both are not applied on



(a) SPECfp2000 (b) NAS

Fig. 7. Performance results of SPECfp2000 and NAS benchmarks

IA-64 for any SPECfp2000 or NAS benchmark. Therefore, they are not used in our experiments.

We have also implemented our techniques in GCC 4.0.0. Note that the part of this analysis for COMMON variables has already been committed in GCC 4.0.0. In GCC 4.0.0, loop unrolling does not split induction variables as it did in GCC 3.5. The lack of such a useful optimization has made our loop unrolling optimization less effective in GCC 4.0.0. (This "performance" bug may be fixed in future GCC releases.)

Figure 7(a) gives the performance results for SPECfp2000. With all our techniques in place, a performance increase of 14.7% is obtained. This result is less impressive compared to our performance achievements in GCC 3.5. The major reason is that loop unrolling that is the most effective in GCC 3.5 has not achieved its full potential due to the performance bug regarding the induction variable splitting we mentioned earlier. However, GCC 3.5 with our improvements incorporated, represented by the "GCC-3.5+OPTS" configuration, outperforms GCC 4.0.0 by 32.5%.

Figure 7(b) shows the performance results for NAS benchmarks. Again, loop unrolling is not as effective as it was in GCC 3.5 for the reason explained earlier. However, our techniques have resulted in a performance increase of 32.0%. Alias analysis and loop unrolling are still the two most effective techniques. Finally, GCC 3.5 with our improvements incorporated, represented by the "GCC-3.5+OPTS" configuration, outperforms GCC 4.0.0 by 48.9%.

6 Related Work

There are a number of open-source compilers for the IA-64 family processors. The Open Research Compiler (ORC) [15] targets only the IA-64 family processors. There are frontends for C, C++ and FORTRAN. The openIMPACT compiler [14] is also designed for the IA-64 architecture alone. Its frontends for C and C++ are not completed yet. One of its design goals is to make openIMPACT fully compatible with GCC.

We have adopted GCC as a compiler platform for this ongoing research because GCC is a multi-language and multi-platform compiler. In addition, GCC is very portable and highly optimizing for a number of architectures. It is more mature than ORC and openIMPACT since ORC and openIMPACT can often fail to compile programs.

There are GNU projects on improving the performance of GCC on IA-64 [16]. However, little results have been included in the latest GCC 4.0.0 version. The SWING modulo scheduler for software pipelining [7] is included in GCC 4.0.0 but does not schedule any loops successfully on IA-64 according to our experimental evaluations.

This work describes four improvements in the current GCC framework for improving the performance of GCC on IA-64. Our improvements are simple but effective in boosting the performance of GCC significantly on IA-64.

Loop unrolling and induction variable optimizations are standard techniques employed in modern compilers [13]. Alias analysis is an important component of an optimizing compiler [2]. In GCC, alias analysis should be carried out not only at both its intermediate representations [6] but also at the frontends for specific programming languages. However, the alias analysis component for FORTRAN programs in GCC is weak. This work demonstrates that a simple intraprocedural alias analysis can improve the performance of FORTRAN programs quite significantly.

Software data prefetching [1,12,17] works by bringing in data from memory well before it is needed by a memory operation. This hides the cache miss latencies for data accesses and thus improves performance. This optimization works the best for programs that have array accesses in which data access patterns are regular. In such cases, it is possible to predict ahead of time the cache lines that need to be brought from memory. Some work has been done on data prefetching for non-array accesses as well [10,11]. Data prefetching represents one of the most effective optimizations in commercial compilers [4,3] for IA-64. For IA-64, adopting rotating register allocation to aid data prefetching in GCC is attractive.

7 Conclusion

In this paper, we describe some progress we have made on improving the floating-point performance of GCC on the IA-64 architecture. Our four improvements are simple but effective. We have implemented our improvements in both GCC 3.5 and GCC 4.0.0. Our experimental results show significant performance increases for both SPECfp2000 and NAS benchmark programs. The part of our alias analysis regarding COMMON variables has been committed in GCC 4.0.0.

Compared to Intel's icc, GCC still falls behind in terms of its performance on IA-64. The following three kinds of optimizations are critical for icc's performance advantages: loop transformations such as loop interchange, loop distribution, loop fusion and loop tiling, software pipelining and interprocedural optimizations. A preliminary implementation for optimizing nested loops for GCC has been developed [5]. However, its loop interchange transformation cannot

even successfully interchange any loops on IA-64. The SWING modulo scheduler for software pipelining has also been incorporated in GCC 4.0.0 [7]. Due to the imprecision of the dependence analysis in GCC 4.0.0, the SWING modulo scheduler can hardly make a successful schedule on IA-64. In GCC 4.0.0, the function inlining remains to be the only interprocedural optimization supported. We plan to make contributions in these areas in future work. We strike, as our long-term goal, to achieve performance on IA-64 comparable to that by commercial compilers while retaining the improved GCC as an open-source, portable, multi-language and multi-platform compiler.

References

1. D. Callahan, K. Kennedy, and A. Porterfield. Software prefetching. *In Proceedings of the Fourth International Conference on Architecture Support for Programming Languages and Operating Systems*, pages 40–52, 1991.
2. S. Debray, R. Muth, and M. Weippert. Alias analysis of executable code. *In The 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 12–24, 1998.
3. G. Doshi, R. Krishnaiyer, and K. Muthukumar. Optimizing software data prefetches with rotating registers. *In Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques*, pages 257–267, September 2001.
4. C. Dulong, R. Krishnaiyer, D. Kulkarni, D. Lavery, W. Li, J. Ng, and D. Sehr. An overview of the intel ia-64 compiler. *Intel Technology Journal*, Q4 1999.
5. David Edelsohn. High-level loop optimizations for gcc. *In Proceedings of the 2004 GCC Developers' Summit*, pages 37–54, 2004.
6. Sanjiv K. Gupta and Naveen Sharma. Alias analysis for intermediate code. *In Proceedings of the GCC Developers' Summit 2003*, pages 71–78, May 2003.
7. Mostafa Hagog. Swing modulo scheduling for gcc. *In Proceedings of the 2004 GCC Developers' Summit*, pages 55–64, 2004.
8. J. Huck, D. Morris, J. Ross, A. Knies, H. Mulder, and R. Zahir. Introducing the ia-64 architecture. *IEEE Mirco*, pages 12–23, Sept-Oct 2000.
9. Intel. *Intel IA-64 Architecture Software Developer's Manual*, volume 1. October 2002.
10. M.H. Lipasti, W.J. Schmidt, S.R. Kunkel, and R.R. Roediger. Spaid:software prefetching in pointer and call intensive environments. *In Proc 28th International Symposium on Micro-architecture*, pages 231–236, Nov 1995.
11. C.K. Luk and T.C. Mowry. Compiler-based prefetching for recursive data structures. *In Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 222–233, September 1996.
12. T.C. Mowry, M.S. Lam, and A. Gupta. Design and evaluation of a compiler algorithm for prefetching. *In Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 62–73, October 1992.
13. Steven S. Muchnick. *Advanced Compiler Design Implementation*. Academic Press, 1997.
14. openIMPACT. http://www.gelato.uiuc.edu.

15. ORC, 2003. http://ipf-orc.sourceforge.net.
16. The GCC Summit Participants. Projects to improve performance on ia-64. *http://www.ia64-linux.org/compilers/gcc_summit.html*, June 2001.
17. V. Santhanam, E. Gornish, and W. Hsu. Data prefetching on the hp pa-8000. *In Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 264–273, June 1997.
18. Richard M. Stallman. *Using and Porting GNU CC*. Free Software Foundation, Inc., 1994.
19. Sverre Jarp. A methodology for using the Itanium 2 performance counters for bottleneck analysis, 2002. http://www.gelato.org/pdf/Performance_counters_final.pdf.
20. Kejia Zhao, Canqun Yang, Lifang Zeng, and Hongbing Luo. *Analyzing and Porting GNU GCC*. Technical Report, National University of Defense Technology, P. R. China, 1997.

An Integrated Partitioning and Scheduling Based Branch Decoupling

Pramod Ramarao and Akhilesh Tyagi

Department of Electrical & Computer Engineering,
Iowa State University, Ames 50011, USA
{pramod, tyagi}@iastate.edu

Abstract. Conditional branch induced control hazards cause significant performance loss in modern out-of-order superscalar processors. Dynamic branch prediction techniques help alleviate the penalties associated with conditional branch instructions. However, branches still constitute one of the main hurdles towards achieving higher ILP. Dynamic branch prediction relies on the temporal locality of and spatial correlations between branches. Branch decoupling is yet another mechanism that exploits the innate lead in the branch schedule with respect to the rest of the computation. The compiler is responsible for generating the two maximally decoupled instruction streams: branch stream and program stream. Our earlier work on trace based evaluation of branch decoupling demonstrates a performance advantage of between 12% to 46% over 2-level branch prediction. However, how much of these gains are achievable through static, compiler driven decoupling is not known. This paper answers the question partially. A novel decoupling algorithm that integrates graph bi-partitioning and scheduling, was deployed in the GNU C compiler to generate a two instruction stream executable. These executables were targeted to branch decoupled architecture simulator with superscalar cores for the branch stream and program stream processors. Simulations show an average performance improvement of 7.7% and 5.5% for integer and floating point benchmarks of the SPEC2000 benchmark suite respectively.

1 Introduction

Branch instructions contribute significantly to our inability to boost the instruction level parallelism. The main incumbent methodology to address this problem is dynamic branch prediction. The trace cache architectures can also soften the control resolution penalty blow. Branch decoupling is yet another technique to resolve branches early. Branch decoupling is predicated on the assumption that the branch schedules have some slack with respect to the schedules of the instructions that depend on them. A branch decoupled architecture is designed to exploit the branch slack to feed the control flow for the program in advance. This is done with the help of a branch decoupled compiler which generates two instruction streams: B-stream for branch stream and P-stream for program stream.

T. Srikanthan et al. (Eds.): ACSAC 2005, LNCS 3740, pp. 252–268, 2005.

The sole responsibility of the B-stream is to resolve branches for P-stream in advance of when P-stream needs them. The compiler's task is to assign instructions to the two streams with the twofold objectives of *(a)* apportioning roughly equal number of instructions to the two streams to balance the load, *(b)* estimate the branch schedules, and to ensure that the coupling points between the two streams still allow the branch of a given basic block to be computed before the corresponding basic block in the P-stream needs it. This paper develops a decoupling algorithm to meet these twin objectives of load balancing and desirable branch favoring schedules. A third issue has to do with the coupling (synchronization) points between the two streams. Each synchronization point creates communication and synchronization overhead. It resets the branch schedule erasing any schedule gains the B-stream might have made until that point. Hence, we need to minimize the synchronizing communication between the two streams. The proposed decoupling algorithm uses a min-cut based bi-partitioning algorithm (Kernighan-Lin) to achieve load-balanced instruction streams with minimal synchronization communication objective. The branch instruction schedules are bootstrapped through interleaved ASAP scheduling phases. The bipartition and scheduling steps feed into each other, converging on a good partition and a good schedule simultaneously.

Note that branch decoupled architectures are not merely a superscalar scheduler prioritized for branch instructions and source instructions for branches. The dynamic instruction window within which the scheduler is able to prioritize branch computations is significantly smaller than the static window available to the compiler. The other difference, of course, is the acceptable complexity of the decoupling algorithms for a compiler versus for a dynamic scheduler. Another point to note is that branch decoupling appears to target a different, potentially orthogonal, attribute of branch instructions than the dynamic branch prediction. The dynamic branch prediction is predicated upon temporal locality of and spatial correlations within branch instructions. The branch decoupling, on the other hand, orchestrates and exploits branch schedule's lead relative to the schedules of branch condition/target consumer instructions. Our earlier work [8] validates this hypothesis.

We developed and implemented an integrated version of Kernighan-Lin bipartition algorithm and ASAP scheduling for branch decoupling. This algorithm was incorporated into the GNU C Compiler. The binaries from this version of the GNU C Compiler are targeted for a two-stream branch decoupled architecture simulator wherein each stream processor is an out-of-order superscalar core. The entire branch decoupled environment was evaluated with SPEC 2000 CPU benchmarks. The base case is dynamic branch prediction through a 2-level predictor. Performance improvements averaging 7.7% for integer benchmarks and 5.5% for floating point benchmarks were observed. We are currently evaluating many other variants of the basic combined bipartition and scheduling driven decoupling.

We describe related work next. Section 3 provides an overview of branch decoupled architectures. The decoupling algorithm for the compiler is described

in Section 4. The experimental methodology is described in Section 5, and the results are presented in Section 6. Section 7 concludes the paper.

2 Related Work

Decoupled access/execute architectures called Memory Decoupled Architectures (MDA) were introduced by Smith in [10]. Here, decoupling is used to isolate instructions that are on the critical path of program execution, (long latency memory instructions in the case of MDA) and execute them on separate execution units of the processor. Decoupling is achieved by splitting the instruction stream into two streams: instructions that access memory and those related to memory accesses are placed in one stream and the other instructions related to general computation are placed in the other stream. Separating instructions which access memory allows data to be prefetched so its available to the other stream with minimum access latency. The MDA contains two processors: an address processor and an execution processor which communicate through architectural queues and have separate register files and instruction caches.

Tyagi [13] proposed the concept of branch decoupled architectures to alleviate the branch penalty effects in instruction-level-parallel processors. The branch decoupling reduces the penalty of control hazards rather than reducing memory access latencies. In [14] Ng studied a different variant called *Dynamic Branch Decoupled Architectures*. Here, decoupling is performed at run-time and requires no compiler support. The advantage of this model is its compatibility to legacy programs. The dynamic branch decoupled processor dynamically decouples programs in a *Decoupling and Fetch Queue* and the resulting two streams are executed by two separate execution units in the processor.

Branch decoupling was evaluated with a trace in [8]. An execution trace is first generated by an initial run of the program by an execution-driven simulator. Simple decoupling algorithms are then applied on the execution trace and the generated trace is re-run on a trace-driven out-of-order simulator. In order to enhance performance, branch prediction is used on both the branch and program processors. The results indicated a speedup of 1.14 to 1.17 in case of floating point benchmarks and a speedup of 1.12 to 1.46 in case of integer benchmarks.

Patt *et al.* [2] used Simultaneous Subordinate Microthreading to improve branch prediction. SSMT architectures spawn multiple concurrent microthreads in support of the primary thread and can be used for a variety of tasks. The idea is to identify *difficult-paths* that frequently mis-predict beyond a threshold to guide microthread prediction. Subordinate threads are constructed dynamically to speculatively pre-compute branch outcomes along frequently misdirected paths. There is no software support and the entire mechanism is implemented completely in hardware. Performance gains of 8.4% on the average (with a maximum of 42%) were obtained over a range of integer and floating point benchmarks.

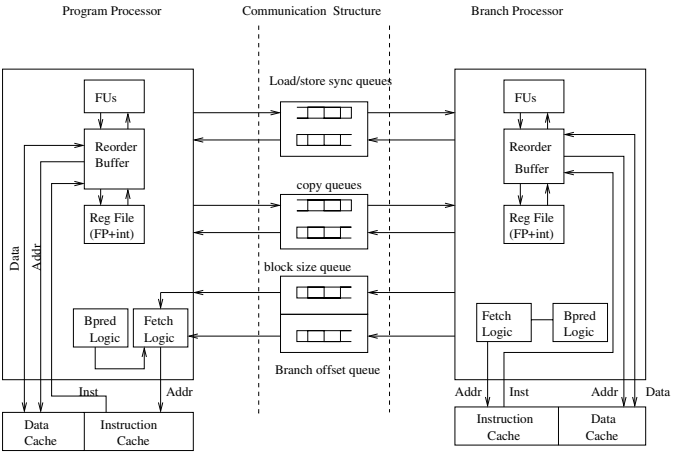


Fig. 1. Branch Decoupled Architecture

3 Branch Decoupled Architecture

Figure 1 shows the schema for the branch decoupled architecture. The two processor cores (branch processor and program processor) are generic superscalar cores. Note that they may equally well be scalar microarchitecture implementations, and that the resources allocated to the two streams, such as dispatch width, can be different.

The three sets of queues in the middle provide the main support for decoupling.

branch condition & offset queues: This constitutes the heart of branch decoupling. The program processor (P-processor) has a non-traditional program counter (PPC). The `PPC` is normally incremented by the fetch size, and a block counter `block-count` is simultaneously decremented by the fetch size each cycle. When the `block-count` reaches zero, the PPC unit dequeues an offset and block-size count from the branch queue. The `PPC` is updated as `PPC` + `offset`, whereas the `block-count` is reset to block-size count retrieved from the branch queue. The block-size count captures the number of instructions to be executed from the next basic block. If the branch queue is empty, the P-processor stalls on it.

copy queues: Some values computed in B-stream are also live in P-stream and vice-versa. The decoupling algorithm can choose to copy a live value from one stream to the other through the appropriate copy queue. The other option, of course, is to let both streams duplicate the entire computation for the live value. The copy queues are designed as the classical producer-consumer data structure. A producer is blocked on a full queue as is a consumer on an empty queue. The producer copying instructions in the producer stream are in one-to-one correspondence with the consumer dequeuing instructions.

Moreover, for program correctness, the relative order of the corresponding copy instructions in the two streams is identical. Hence the copy instructions also act as synchronization barriers.

load/store synchronization queues: The LSQs in each of the B- and P-processors are responsible for maintaining memory consistency. However, the larger, original, one-stream program memory consistency can only be maintained if the two LSQs are synchronized in some cases. Specifically, for each store assigned to B-stream, all the following loads in the one-stream program that get assigned to P-stream should either not be aliased to the store or should be synchronized with respect to it. The load/store synchronization queues are used for such synchronization. If the compiler can ascertain that no aliasing exists between an earlier store and a load assigned to the other stream, no synchronization primitive is inserted. Otherwise, the store and load pair in different streams synchronize through the corresponding load/store queue. A synchronized store deposits a producer token in the queue, and a synchronized load needs to consume that token before proceeding (it is a blocking point for the consumer load). Note that the relative order of the producers in one stream needs to match the corresponding consumer order in the other stream for the synchronization to work.

Decoupling Example: Figure 2 illustrates branch decoupling of a program. Consider the the high-level language source code in Figure 2 top-right. The corresponding assembly translation is shown in Figure 2 top-left box. Assume the registers $2, $3, $4, $5 hold the values of i, a, b, c respectively.

The instruction stream is decoupled into the two streams as shown in Figure 2. The conditional branch instruction, *bltz* is decoupled into the B-stream. Interestingly, there is no corresponding *bltz* in the corresponding basic block in the P-stream. It is not that the P-stream blocks are orphaned, it is just that their control flow is all delegated to the hardware and B-stream. The branch instructions in B-stream need to carry control information about the control flow of the P-stream. A typical branch instruction in B-stream has the format: `bltz $2, $BL1, $PL1, block-size-not-taken, block-size-taken`. The first two arguments (`$2` and `$BL1`) control the flow within B-stream in the traditional way. one is the comparison source operand, and the other is the branch offset specification within B-stream. In addition, however, this instruction needs to know the P-stream offset for both taken and not-taken branch. The not-taken branch offset is 0 and hence is not included in the instruction. The parameter `$PL1` denotes the P-stream taken offset. The parameter `block-size-not-taken` captures the size of the block in P-stream that is executed on a not-taken branch (in number of instructions, could have been bytes). Similarly, `block-size-taken` is the number of instructions in the P-stream block reached on a taken branch. When the branch instruction (*bltz*) is executed by the B-processor, its actions include the following in addition to managing the B-stream control flow through B-processor PC updates. If it is a taken branch, it queues `P-stream-taken-offset` denoted by $PL1 or 2 instructions into the branch offset queue. An offset of 0 is queued for a not-taken branch. Moreover, `block-size-taken` is queued into the block-size

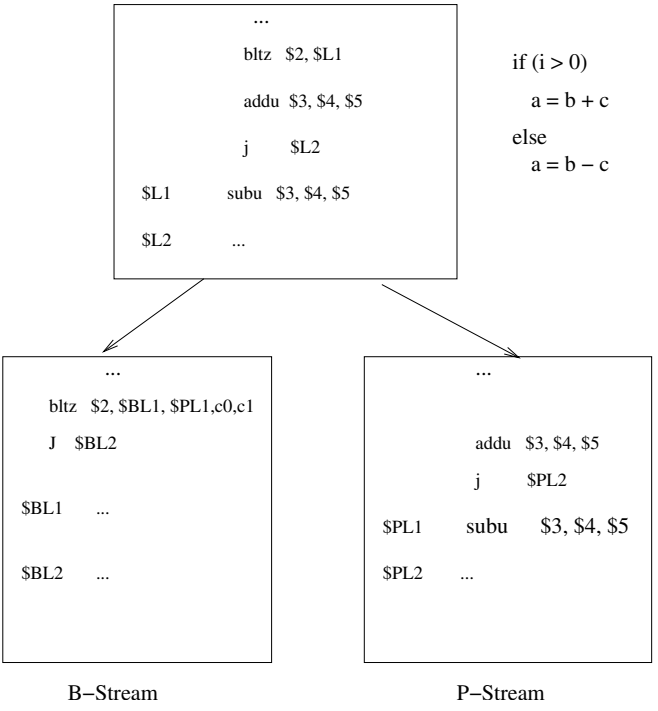


Fig. 2. An Example of Branch Decoupling

queue of the branch queue for a taken branch; whereas `block-size-not-taken` is queued for a not-taken branch.

4 Branch Decoupling

4.1 Branch Decouplability of a Program

It is an intriguing question if a viable metric of decouplability of a program can be developed. Some attributes of such a metric can be characterized as follows. If the decoupling objective was to minimize the communication between the B- and P- streams, then a mincut partition of the program data flow graph can be used to capture branch decouplability.

Definition 1 (branch decouplability – mincut based:). *Let a mincut of a program P's dataflow graph $G = (V, E)$ be of size k. Branch decouplability of P $\alpha(P)$ is defined as* $(1 - (2k/|V|))$.

The mincut is the number of edges across the partition. A program that can partitioned into two cleanly separate streams with no communication has decouplability of 1 (with mincut size equal to 0). The other extreme will be a

complete graph $G = (V, E)$ which cannot occur for a real program. This is because the indegree of each node (corresponding to an instruction) is at most two (corresponding to the two operands).

This definition penalizes every communication edge between the two instruction streams. There is no influence of the instruction schedules on decouplability. In reality, if the schedule of the destination instruction of an edge in the cut is not constrained by this edge, this edge does not really constrain decouplability (separation of the two streams). This occurs if this cut edge has some slack. The slack of an edge (u, v) is given by $(schedule(v) - schedule(u) - 1)$ where $schedule(u)$ is the time when the instruction u is scheduled. Note that whether a cut edge is oriented from B-stream to P-stream or vice versa, it should not constrain the decouplability if it has a positive slack. This leads to the following, somewhat better, definition of decouplability.

Definition 2 (branch decouplability – mincut & schedule based:). *Let C be a mincut of a program P's dataflow graph $G = (V, E)$. Branch decouplability of P $\beta(P)$ is defined as* $(1 - (2 * |\{e \in C \ s.t. \ slack(e) \leq 0\}|/|V|))$.

This definition counts only those edges e in the cut C whose slack is not positive as decoupling constraining. Sections 4.4 and 4.5 describe our decoupling methodology that attempts to minimize the decoupling metric $\beta(P)$.

4.2 Branch Decoupling Compiler

The Branch Decoupling Compiler is one of the most important components of the BDA paradigm. A compiler that is capable of performing decoupling is essential to the operation of the BDA. The branch decoupling compiler (bdcc) is based on the GNU C Compiler [11] that is included in the SS tool set. It is augmented with control-flow, data-flow, dependence and alias analyses and a decoupling algorithm. The decoupling process in the compiler usually takes place as the last pass before assembly code generation. The compiler uses an interleaved text segment interpretation for the purpose of applying traditional compiler techniques. The final binary executable generated however consists of two text segments for each processor.

4.3 Decoupling Pass Design

This section provides an outline of the decoupling pass that is implemented in the compiler. The decoupling pass is applied as the last pass of compilation, before the assembly is generated for the target machine architecture. The decoupling pass is applied on the program, one function at a time and the assembly for the function using the RTL to assembler generation capability in the compiler.

The RTL to assembler code generator in GCC is augmented with an annotation generator which annotates each RTL instruction with the necessary implicit copies and synchronization points as it is converted to assembler output.

The decoupling pass as implemented by the compiler is outlined below. Each step of the algorithm is implemented as a function in the compiler which performs the necessary task. Decoupling is performed in three phases. In the first phase, the basic blocks are determined, DAGs and the control-flow graph are constructed. In the second phase, decoupling based on interleaved KL heuristic and scheduling is applied. In the last phase, data-flow analysis is performed to propagate dependences, and the necessary copy instructions and synchronization points are inserted into the instruction stream.

1. *find_basic_blocks* : Analyze RTL chain of the function and identify basic blocks.
2. *construct_dags* : For each basic block, using dependence analysis, construct DAGs.
3. *construct_flowgraph* : Using the RTL chain of the function and basic block information, identify *successors* and *predecessors* of each basic block to generate the control flow graph.
4. *kl_decouple* : For each basic block
   (a) $change = 0$
   (b) *init_cost_matrix* : Determine critical path of the basic block using single source shortest path. Compute slacks of each instruction and initialize cost matrix.
   (c) *init_kl* : Initialize partitions A and B for the two streams using critical path information and insert dummy nodes if necessary. Apply KL heuristic to the DAG.
   (d) *schedule* : Apply scheduling to the basic block.
   (e) If change in partitions, $change = 1$.
   (f) If $change = 1$ goto step a.
   (g) Label the RTL instructions as belonging to either P stream or B stream according to the partitions obtained.
5. *propagate* : Apply data-flow analysis to propagate dependences beyond basic blocks.
6. *decouple_2* : Using data-flow information, insert copy information into the RTL structure for each instruction.
7. *final_decoupling* : Apply dependence analysis using memory alias analysis to determine synchronization points in the instruction stream. Add this to the RTL structure for each instruction.

The partitioning and scheduling technique is described in greater detail in the subsequent sections.

4.4 Decoupling Process

Branch decoupling can be viewed as a graph bi-partitioning problem where the graph corresponds to the DAG of a basic block or the DAG corresponding to a particular control-flow path of the program. The objective of decoupling can then be thought of as an attempt to find a partition of the instruction stream into two

streams - the *B stream* and the *P stream* with a second important objective of minimizing the communication between the two streams. As described before, this is beneficial as it allows the two streams to execute as independently as possible without one having to stall while waiting for a value from the other.

This section presents a branch decoupling algorithm based on a technique which utilizes a combination of graph bi-partitioning and scheduling. We propose to use the Kernighan-Lin iterative algorithm for graph bi-partitioning.

The Kernighan-Lin algorithm [6] [7] is a specialized simulated annealing algorithm for solving the graph partitioning problem. It modifies search procedure to allow the system to escape the local minimum unlike some simple greedy algorithms. The search strategy chooses the vertex pair whose exchange results in the largest decrease or the smallest increase if no decrease is possible. In order to understand the algorithm, some definitions are first introduced.

The algorithm starts with an edge weighted graph

$$G = (V, E, W_E)$$

and an initial partition

$$V = A \cup B, |A| = |B|$$

The KL algorithm attempts to find two partitions, X and Y such that $X \subset A$, $Y \subset B$. The algorithms proceeds by swapping nodes between X and Y with the objective of minimizing the number of external edges connecting the two partitions. Each edge is weighted by a cost and the objective corresponds to minimizing a cost function called the total external cost, or cut weight W.

$$CurrentWeight, W = \sum_{a \in A, b \in B} w(a, b), \tag{1}$$

where *w(a,b)* is the weight of edge (a,b).

If the sets of vertices X and Y are swapped, $A_{new} = (A - X) \cup Y, B_{new} = (B - Y) \cup X$, then the new cut weight W_{new} is given by

$$NewWeight, W_{new} = \sum_{a \in A_{new}, b \in B_{new}} w(a, b), \tag{2}$$

In order to simplify the measurement of the change in cut weight when nodes are interchanged, *external* and *internal* edge costs are introduced. For every $a \in A$, the following is maintained

E(a) = external cost of $a = \sum_{b \in B} w(a, b)$

I(a) = internal cost of $a = \sum_{á \in A, á \neq a} w(a, á)$

The cost difference is the difference between the external edge costs and internal edge costs,

D(a) = E(a) - I(a)

Similarly for every $b \in B$, the same information is maintained

E(b) = external cost of $b = \sum_{a \in A} w(a, b)$

I(b) = internal cost of $b = \sum_{b́ \in B, b́ \neq b} w(b, b́)$ and

D(b) = E(b) - I(b)

If any vertex pair (a,b) is picked from A and B respectively and swapped, the reduction in cut weight is called the *Gain, g*. This can be expressed as

$$Gain(a,b) = I(a) - (E(a) - w(a,b)) + I(b) - (E(b) - w(a,b))$$

$$Gain(a,b) = D(a) + D(b) - 2w(a,b)$$

After the vertices are swapped, the new D values are computed by
$D(\acute{x}) = D(x) + 2w(x,a) - 2w(x,b), x \in A - a$
$D(\acute{y}) = D(y) + 2w(y,b) - 2w(y,a), y \in B - b$

The KL algorithm finds a group of node pairs to swap that increases the gain even though swapping individual node pairs from that group might decrease the gain. Some of the terms outlined above are shown in Figure 3.

The algorithm is outlined below

1. Initialize partitions A & B and compute total weight, W.
2. Compute D(x) for all vertices x.
3. Unlock all vertices. Set $i = 1$.
4. While there are unlocked vertices do
   (a) Find the unlocked pair (a_i, b_i) that maximizes $Gain(a_i, b_i)$.
   (b) Mark a_i and b_i (but do not swap).
   (c) Update D(x) for all unlocked vertices x, pretending that $a_i \& b_i$ have been swapped.
   (d) $i \leftarrow i + 1$.
5. Pick j that maximizes $Gain = \sum_{i=1}^{j} Gain(a_i, b_i)$
6. If $Gain > 0$ then update

$$A = (A - \{a_1, a_j\}) \cup \{b_1, \ldots b_j\},$$

$$B = (B - \{b_1, \ldots b_j\}) \cup \{a_1, \ldots a_j\},$$

$$Weight = Weight - Gain$$

7. If $Gain > 0$, go to step 2.

Step 4 is executed $|V|/2$ times during each iteration while step 4(a) requires $O(|V|^2)$ time. If the number of iterations is a fixed constant, the total running time of the KL algorithm is $O(|V|^3)$. In the algorithm, the $Gain(a_i, b_i)$ may be negative and this allows the algorithm to escape some local minima.

Since the KL algorithm is heuristic, only a single iteration may result in a local optimum which may not be the global optimum. The heuristic is repeated starting with the new bisection. The algorithm usually terminates in at most five iterations. Another property of the KL heuristic is that handles only exact bisections of graphs. This restriction is eliminated by adding dummy vertices that are isolated from other vertices before the algorithm is applied. An example of the KL partitioning algorithm as applied to the graph in Figure 3 is shown in Figure 4.

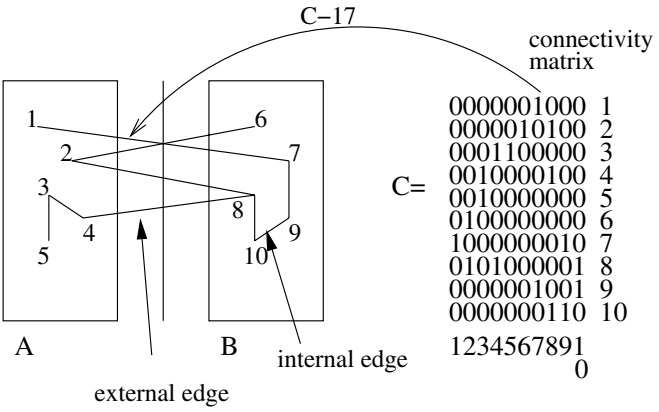


Fig. 3. Terms used in KL algorithm

4.5 Cost Function and Slack Analysis

Since the KL heuristic requires a graph with edges weighted by a cost value and a good initial partition to ensure early termination, a methodology is needed to assign proper cost values and determine an initial partition before the algorithm is executed. This section presents a brief overview of the methodology of assigning cost weights to edges that is implemented in the compiler.

The cost matrix for the DAG is computed based on *slacks* [5] of the instructions and the critical path length of the DAG. The cost of each edge $w(a, b)$ is based on the cost function

$$w(a,b) = \frac{CP_{length}}{Min(Slack_{op_1}, Slack_{op_2})} \tag{3}$$

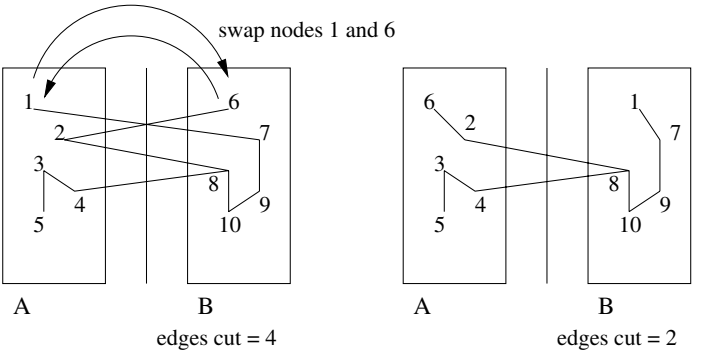


Fig. 4. Partitioning using KL algorithm

where CP_{length} is the critical path length of the DAG and op_1 and op_2 are the two input operands of the instruction. $Slack_{op_1}$ and $Slack_{op_2}$ are the slacks of the instructions which produce op_1 and op_2 respectively.

In order to compute the critical path, the DAG is first topologically sorted. This involves performing a *depth first search* and placing each completely visited vertex at the head of a linked list. Weights are assigned to each edge. These weights are based on the latencies of each instruction in the DAG as read from the target machine description file in the compiler. The weights are negated and the single source shortest path algorithm [3] is applied to generate the critical path length. The instructions present on the critical path are also stored.

The instructions (and the terminating branch instruction) that are on the critical path in the DAG are assigned to the *B stream*. All the remaining instructions are assigned to the *P stream*. This ensures that branch conditions are given a priority to be evaluated early by the B stream. This forms the initial partition to the KL algorithm.

In order to determine the slacks of the instructions, the notion of instruction slacks is adopted from [5]. The slack of each node (instruction) is determined by computing the slack of each edge in the DAG. The slack of each edge $e = u \rightarrow v$ is the number of cycles that the latency of e can tolerate without delaying the execution of target node v. This is computed as the difference between the arrival time of the last-arriving edge sinking on v and the arrival time of e. The slack of a node v is the smallest local slack of all the outgoing edges of v.

The cost matrix of each DAG is determined using the methodology described above before branch decoupling by the KL algorithm is applied by the compiler. All the instructions on the critical path are assigned a cost of value equal to the critical path length. The remaining instructions are assigned cost weights according to Equation 3. The reason behind the cost function in Equation 3 is that instructions on the critical path (including the terminating branch instruction) must be placed in the B stream as explained above. Thus higher cost weights are assigned to the outgoing edges of instructions on the critical path and lower weights according to Equation 3 are assigned to the outgoing edges of the remaining instructions. This ensures that the KL heuristic would attempt to place as many critical instructions as possible in the B stream itself.

4.6 Scheduling

The DAG prescribes the dependences between the instructions, but scheduling needs to be applied to determine the start times of each instruction. After the KL-heuristic is applied, scheduling is performed on both the streams. In certain cases, the instructions in one stream may not be dependent on each other but may be dependent on instructions in the other stream. This happens as decoupling using KL heuristic attempts to swap every vertex pair to determine an optimal partition. As a result, scheduling becomes necessary to determine whether a better partition can be found.

The scheduling algorithm used in branch decoupling is the simple *ASAP* (As Soon As Possible) unconstrained scheduling algorithm [4]. This algorithm

proceeds by topologically sorting the nodes of the graph. Scheduling of instructions $\{v_0, v_1, \ldots, v_n\}$ in the sorted order starts with the instruction v_0. Then an instruction whose predecessors are already scheduled is next selected to be scheduled. This procedure is repeated until every instruction is scheduled. The algorithm is *ASAP* in the sense that the start time for each operation is the least one allowed by the dependences.

Let $G_s(V, E)$ be the DAG to be scheduled. The algorithm is presented below

1. Schedule v_0 by setting time $t_0 = 1$.
2. repeat
   (a) Select a vertex v_i whose predecessors are already scheduled.
   (b) Schedule v_i and set $t_i^S = max_{j:(v_j,v_i)\in E}(t_j^S + d_j)$
3. until v_n is scheduled.

5 Evaluation Methodology

5.1 Branch Decoupled Architecture Simulation Tool Set

In order to simulate the BDA, a BDA Simulation Tool set derived from the SimpleScalar simulation tool set [1] was developed. The tool set is outlined in Figure 5.

The BDA simulation tool set consists of the following components:

1. **bdcc** - This is the branch decoupling compiler based on the GNU C compiler as explained in the previous sections.
2. **sim-bdcorder** - This is a true out-of-order execution simulator that has been designed based on *sim-outorder* that is available with the SimpleScalar tool set. *sim-outorder* performs execution of instructions in the dispatch stage itself and then builds timing information to give an effect of out-of-order execution. But *sim-bdcorder* performs execution of instructions out-of-order thus simulating a true out-of-order superscalar processor. Since *sim-outorder* performs execution of instructions in the dispatch stage rather than in between the issue and writeback stages, it requires the values in the register file to be available to an instruction at dispatch. This creates a few problems for the BDA since the BDA follows a dual-processor model. If an instruction needs to wait on a value that is to be communicated by the other stream (a copy), it will stall dispatch unnecessarily. *sim-bdcorder* does actual execution of instructions in the issue stage and works with RUU (register update unit) entries rather than with the register file. The *sim-bdcorder* simulator (with only a single processor active) was tested against the *sim-outorder* simulator to validate that identical results are obtained for simulation runs of benchmark programs.
3. **binary utilities** - The BDA simulation tool set includes various binary utilities (assembler, linker) that have been retargeted to the BDA platform. In addition, a utility is included which generates the two text segment binary executables for the BDA architecture.

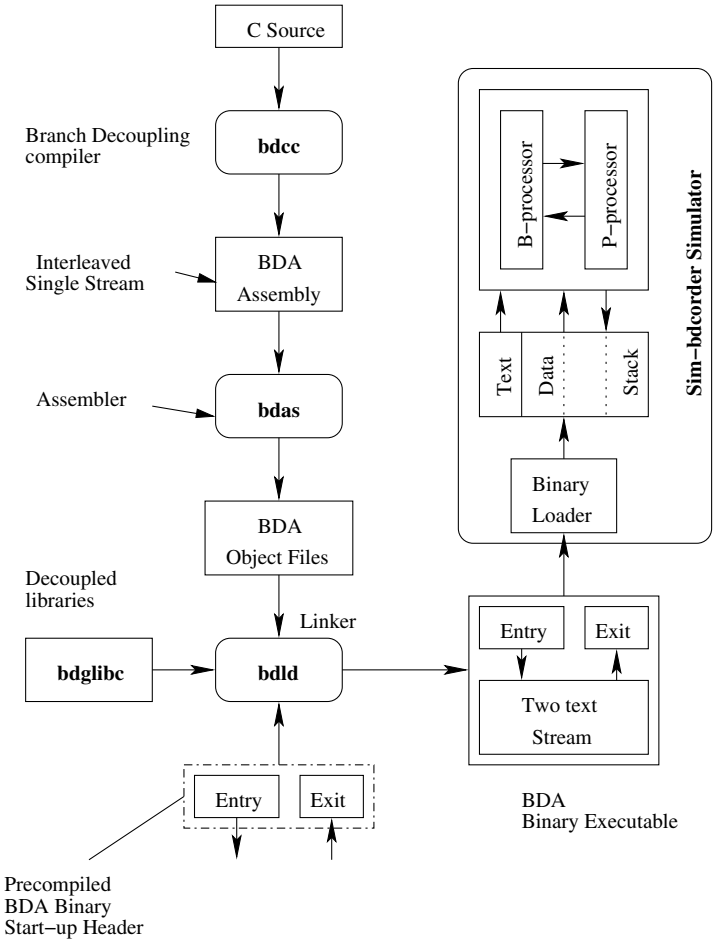


Fig. 5. BDA Simulation Tool set

4. **bdglibc** - The *glibc* libraries that are available with SimpleScalar have been retargeted to work with the BDA platform. It includes a retargeted precompiled binary start-up header that initializes program execution.

5.2 Methodology

The simulation environment of the branch decoupled architecture consists of the toolchain described in the previous section.

A set of integer and floating point benchmark programs are adopted from the SPEC CPU2000 benchmark suite [12]. The benchmark programs are first compiled with the branch decoupling compiler (with optimization level O3) presented in Section 4.2. The decoupling compiler applies the branch decoupling

algorithm on the benchmark program and uses the retargeted toolchain to produce decoupled binary executables. The resulting binaries are then simulated by the out-of-order execution BDA simulator.

Table 1. Number of instructions executed and amount skipped

Program	#Executed (millions)	#Skipped (millions)
art	500	1500
equake	500	1500
bzip2	500	200
gcc	500	1000
gzip	500	40
mcf	500	1500
parser	500	250
twolf	500	400

Table 2. Architectural parameters

Parameter	Value
Issue	4-way Out-of-order
Fetch Queue Size	32 instructions
Branch Prediction	2K entry bimodal
Branch mis-prediction latency	3 cycles
Instruction Queue Size (RUU)	128 instructions
Load/Store Queue Size (LSQ)	8 instructions
Integer Functional Units	4 ALUs, 1 Mult./Div.
Floating Point Functional Units	4 ALUs, 1 Mult./Div.
L1 D- and I-cache	Each: 128Kb, 4-way
Combined L2 cache	1Mb, 4-way
L2 cache hit latency	20 cycles
Main memory hit time	100 cycles

During the simulation runs, execution of each benchmark is forwarded by a particular amount of instructions and then execution is simulated for 500 million instructions. The number of instructions executed and the number of instructions skipped for each benchmark are based on [9]. The benchmarks and the number of instructions skipped in each case are outlined in Table 1. In order to get base processor simulation results, the benchmarks are compiled with the GNU C compiler that comes with the SimpleScalar tool set (with optimization level 03) and are simulated using the *sim-outorder* simulator.

The system model on which each of the processors within *sim-bdcorder* is based on a typical out-of-order superscalar processor. Table 2 contains a description of the baseline architectural parameters.

6 Experimental Results

In this section, some performance results of the BDA in comparison with the base processor model are presented.

The resulting IPC (Instructions per Cycle) for both the system models are shown in Figure 6. Figure 7 shows the percentage of the time a branch condition is available in the branch queue for the P processor when the queue is accessed by it. This gives an indication of the percentage of the time the P processor stalls while waiting for the branch outcome to by computed by the B processor.

twolf and *parser* show the best speedups. The branch conditions are available for both these benchmarks about 90% of the time. The performance of both benchmarks improve by 17.5% and 15% respectively. However, *gcc* and *mcf* show performance degradation by 6% and 2% respectively. It can be seen from Figure 7 that the branch outcomes are available only 65% and 67% of the time respectively

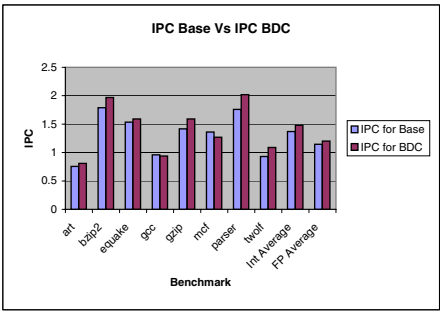


Fig. 6. IPC comparison between the base and BDA processors

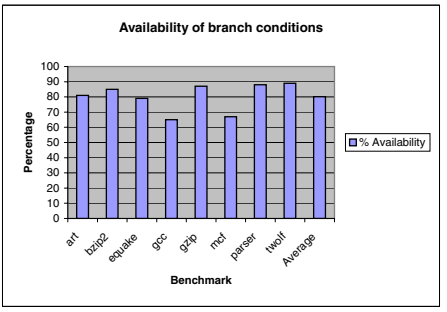


Fig. 7. Availability of branch conditions when required

which explains the degradation in performance. The performance improvements for integer benchmarks is about 7.7% on the average. The BDA performs well on floating point benchmarks with about 5.5% performance improvement on the average.

7 Conclusion and Future Work

Branch Decoupled Architectures offer a new paradigm to help alleviate the penalties associated with conditional branches in modern processors. This paper presented an overview of branch decoupled architectures and a novel decoupling algorithm that is used by a compiler to perform branch decoupling. This algorithm is based on a combination of graph bi-partitioning and scheduling to achieve maximal decoupling. It also presented a toolchain that has been retargeted to the BDA platform to help study the various performance characteristics. There are many possible variations of the presented decoupling algorithm that emphasize mincut and scheduling aspects differently. We are exploring such variations. Another important aspect is capturing the decouplability of a program with a simple attribute. We have made an attempt at a preliminary definition of decouplability in this paper. However more work needs to be done in this direction.

References

1. D. Burger and T. Austin. The simplescalar tool set, ver 2.0. *Technical Report 1342 Computer Science Department, University of Wisconsin-Madison*, 1997.
2. R. Chappell, F. Tseng, Y. Patt, and A. Yoaz. Difficult-path branch prediction using subordinate microthreads. *Proceedings of IEEE 29th Annual International Symposium on Computer Architecture (ISCA)*, 2002.
3. T. Cormen, C. Leiserson, and R. Rivest. Introduction to algorithms, 2nd ed. *MIT Press*, 1990.

4. G. De Micheli. Synthesis and optimization of digital circuits. *McGraw Hill, Inc.*, 1994.
5. B. Fields, R. Bodik, and M. Hill. Slack: Maximizing performance under technological constraints. *Proceedings of IEEE International Symposium on Computer Architecture (ISCA)*, 2002.
6. B. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *The Bell System Technical Journal*, 1970.
7. T. Lengauer. Combinatorial algorithms for integrated circuit layout. *John Wiley and Sons*, 1990.
8. A. Nadkarni and A. Tyagi. A trace based evaluation of speculative branch decoupled architectures. *Proceedings of IEEE International Conference on Computer Design (ICCD)*, 2000.
9. S. Sair and M. Charney. Memory behavior of the spec2000 benchmark suite. *Technical Report RC-21852, IBM T.J. Watson Research Center*, 2000.
10. J. Smith, S. Weiss, and N. Pang. A simulation study of decoupled architecture computers. *IEEE Transactions on Computers*, 1986.
11. R. Stallman. Gnu compiler collection internals. *Free Software Foundation*, 2002.
12. Standard Performance and Evaluation Corporation. SPEC CPU2000 V1.2. 2000.
13. A. Tyagi. Branch decoupled architectures. *Proceedings of the Workshop on Interaction between Compilers and Computer Architecture, 3rd International Symposium on High-Performance Computer Architecture*, 1997.
14. A. Tyagi, H. Ng, and P. Mohapatra. Dynamic branch decoupled architecture. *Proceedings of the IEEE International Conference on Computer Design*, 1999.

A Register Allocation Framework for Banked Register Files with Access Constraints

Feng Zhou[1,2], Junchao Zhang[1], Chengyong Wu[1], and Zhaoqing Zhang[1]

[1] Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China
{zhoufeng, jczhang, cwu, zqzhang}@ict.ac.cn
[2] Graduate School of the Chinese Academy of Sciences, Beijing, China

Abstract. Banked register file has been proposed to reduce die area, power consumption, and access time. Some embedded processors, e.g. Intel's IXP network processors, adopt this organization. However, they expose some access constraints in ISA, which complicates the design of register allocation. In this paper, we present a register allocation framework for banked register files with access constraints for the IXP network processors. Our approach relies on the estimation of the costs and benefits of assigning a virtual register to a specific bank, as well as that of splitting it into multiple banks via copy instructions. We make the decision of bank assignment or live range splitting based on analysis of these costs and benefits. Compared to previous works, our framework can better balance the register pressure among multiple banks and improve the performance of typical network applications.

1 Introduction

Network processors have been widely adopted as a flexible and cost-efficient solution in building today's network processing systems. To meet the challenging functionality and performance requirements of network applications, network processors incorporate some unconventional, irregular architectural features, e.g. multiple heterogeneous processing cores with hardware multithreading, exposed memory hierarchy, banked register files, etc. [1]. These features present new challenges for optimizing compilers.

Instead of building a monolithic register file, banked register file has been proposed to provide the same bandwidth with fewer read/write ports[2]. The reduction of number of ports lowers the complexity of the register file, which further leads to the reduction of die area, power consumption, and access time[3][4]. For banked register files, however, there may be conflicts when the number of simultaneous accesses to a single bank exceeds the number of ports of the bank. While superscalar designs typically solve bank conflicts with additional logic, embedded processors mostly left the problem to programmer or compiler via exposing some access constraints in ISA, therefore simplify the hardware design. For compiler, this complicates the problem of register allocation since in addition to the interferences between two virtual registers, there may be bank conflicts between their uses which may further limit the allocation of registers to them.

T. Srikanthan et al. (Eds.): ACSAC 2005, LNCS 3740, pp. 269–280, 2005.

In this paper, we present a register allocation framework for banked register files with access constraints for the IXP network processors [5][6]. Our approach relies on estimation of the costs and benefits of assigning a virtual register to a specific bank, as well as that of splitting it into multiple banks via copy instructions. We make the decision of bank assignment or live range splitting based on analysis of these costs and benefits. This helps to balance the register pressures among the banks. When splitting a live range, we use copy instructions instead of loads/stores and force the split live ranges to different banks. Though this may introduce additional copies, it can reduce the number of memory accesses significantly.

The rest of this paper is organized as follows. Section 2 introduces some related architectural features of the IXP network processor, especially its banked organization of the register files and the access constraints. Section 3 describes the compilation flow and the proposed register allocation framework. It also provides further details on cost and benefit estimation and live range splitting. Section 4 presents the experimental results. Section 5 describes related works, and section 6 concludes the paper.

2 IXP Architecture and Register File Organization

The IXP network processor family [5] was designed as core processing component for a wide range of network equipments including multi-service switches, routers, etc. It's a heterogeneous chip multiprocessor consisting of an XScale processor core and an array of MicroEngines (MEs). The XScale core is used mainly for control path processing while the MEs for data path processing. IXP has a multi-level, exposed memory hierarchy, consisting of local memory, scratchpad memory, SRAM, and DRAM. Each ME has its own local memory, while scratchpad memory, SRAM, and DRAM are shared by all MEs. The stack was implemented using both local memory and SRAM, starting from local memory and growing into SRAM. The MEs have hardware support for multi-threading to hide the latency of memory accesses.

To handle the large amount of packet data and to service the multiple threads, IXP provides several large register files. Figure 1 is a diagram of MEs register files. Each ME has four register files: a general purpose register file (GPR) which are mainly used by ALU operations, two transfer register files for exchanging data with memory and other I/O devices, and a next neighbor register file for efficient communications between MEs.

These register files, except next neighbor, are all partitioned into banks. The GPR is divided into two banks: GPR A and GPR B, each have 128 registers. The transfer register files are partitioned into four banks: SRAM transfer in, SRAM transfer out, DRAM transfer in, and DRAM transfer out.

ME instructions can have two register source operands. We refer to them as A and B operand. Therere some restrictions, which are called *"two source-operand selection rule"* in [7], on where the two operands of an instruction can come from. They are summarized as follows [7]:

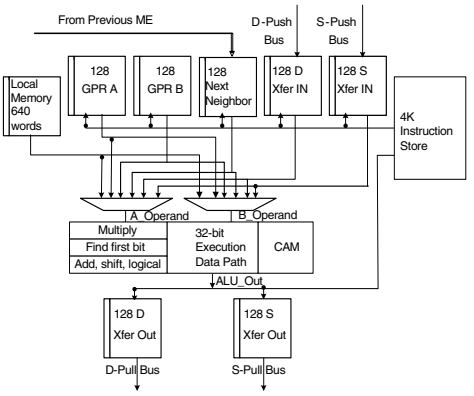


Fig. 1. IXP 2400 Register File Organization

- An instruction can't use the same register bank for both A and B operands.
- An instruction can't use any of SRAM Transfer, DRAM Transfer and Next Neighbor as both A and B operands.
- An instruction can't use immediate as both A and B operands.

If an instruction does not conform to the constraints listed above, we say the instruction has *"bank conflict"* and it's a *"conflict instruction"*. This puts new challenges to register allocation since it has to deal with the bank assignment for each virtual register. In this paper, we focus on GPR's bank conflict problem but the proposed technique applied to other register classes as well.

3 A Register Allocation Framework Solving Bank Conflicts

We designed a register allocation framework as part of the Shangri-La infrastructure, which is a programming environment for IXP [8]. Shangri-La encompasses a domain-specific programming language named `Baker` for packet processing applications, a compilation system that automatically restructures and optimizes the applications to keep the IXP running at line speed, and a runtime system that performs resource management and runtime adaptation. The compilation system consists of three components: the profiler, the pipeline compiler, and the aggregate compiler. The work presented here is part of the aggregate compiler, which takes aggregate definitions and memory mappings from pipeline compiler and generates optimized code for each of the target processing cores (e.g. MicroEngines and XScale). It also performs machine dependent and independent optimizations, as well as domain-specific transformations to maximize the throughput of the aggregates. Figure 2 illustrates the compilation flow of Shangri-La and highlights some phases related to register allocation.

Our register allocation framework is based on the priority-based coloring approach [6] in that the virtual registers are processed in the priority order with

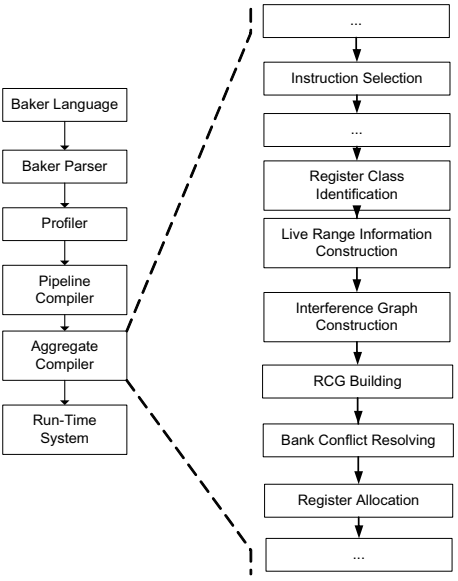


Fig. 2. Shangri-La Compilation Flow

the priorities computed in the same way as described in [6]. As illustrated in Fig. 2, we first perform instruction selection. Then in the phase of register class identification, we analyze each instruction to identify the register classes/files that each symbolic register could reside in. We then build the live ranges and the interference graph. These information are needed for both bank conflicts resolving and register allocation. To resolve bank conflicts, we first build a register conflict graph (RCG) [9]. RCG is an undirected graph where the nodes represent the virtual registers and an edge indicates that two virtual register can not be assigned to the same register bank. Based on the RCG, we assign the register bank to each virtual register (algorithm shown in Fig. 3). For each virtual register, we estimate the costs and benefits of assigning it to a specific bank. We also estimate the cost of splitting it into multiple banks via copy instructions, which is the total cost of the generated sub-liveranges plus the cost of the inserted inter-bank copy operations. The cost of a sub-liverange is the minimum of the costs of assigning it to bank A or bank B and can be estimated through invoking the EstimateCost function recursively. However, we limit the depth of recursion with a small threshold here. We then determine bank assignment for the virtual register based on analysis of these costs. The AssignRegBank function assigns the current node to the given bank, mark it as spill if needed and update the *CountOfSpill* variable.

When all virtual registers have been assigned banks in this way, some instructions may have both operands been assigned to a same register bank, causing bank conflicts. We resolve these conflicts by inserting inter-bank copy instructions before each conflicting instruction. E.g. for an instruction `r=op(a,b)` in

```
1: procedure BankConfResolving(RegisterConflictGraph)
2:   CountOfSpills = 0  // the number of nodes that have been marked spill so far
3:   for all node v of RegisterConflictGraph do
4:     CostA = EstimateCost(v, BANK A) // Calculate the cost of assigning v to GPR A
5:     CostB = EstimateCost(v, BANK B)
6:     SplitCost = SplitCost(v)
7:     if SplitCost is minimum in these three costs then
8:       SplitNode(v)
9:     else if CostA >= CostB then
10:      AssignRegBank(v,BANK B)
11:    else
12:      AssignRegBank(v,BANK A)
13:    end if
14:  end for
```

Fig. 3. Resolving Bank Conflicts

which `a` and `b` are the two register source operands and assigned to a same bank, we first introduce a new virtual register `c` and insert `c=a` before this instruction, then we assign `c` to the other bank than the one assigned to `b` and rename a in `r=op(a,b)` to `c`. This guarantees that all conflicts are resolved. The traditional register allocation phase follows to allocate registers per bank for all virtual registers per bank.

3.1 Cost and Benefit Analysis

Bank assignment decisions are based on analysis of the costs and benefits of assigning a virtual register to a specific register bank. The compiler can then use these results to trade-off between the two banks. Below we describe how the cost-benefit estimation function calculates the impact based on the following factors:

Conflict-Resolving Cost: Bank conflicts between two virtual registers need to be resolved through inserting copies before conflict instructions. The copy operation costs one cycle, so the total number of inserted copy represents the conflict-resolving cost. The ConflictResolvingCost function shown in Fig. 4 computes the number of instructions that use both the given virtual register and a virtual register that conflicts with it on RCG.

Spill Cost: Though each ME has 256 GPR on it, each thread has only 32 GPRs (in two banks) when ME runs in 8-threads mode. Assigning too many virtual registers to a single bank may cause spills in that bank while leaving the other bank underutilized. Balancing the register pressure between two banks is an important consideration in our framework. The SpillCost function estimates this cost for a virtual register. We first check the number of live ranges that have higher priorities and being assigned the same register bank. If the number is larger than the number of allocatable register, we treat it as going to be spilled and compute the corresponding spill/reload cost. Otherwise, the spill cost is zero.

Coalescing Benefit: The source and result operand of copy instructions can reside in any bank if they are both of GPR type. If they reside in the same bank, latter phases (e.g. register coalescing [10]) may have an opportunity to

```
 1: procedure EstimateCost(vr,regbank)
 2:   SpillCost        = SpillCost(vr,regbank)
 3:   ConfResolvCost   = ConflictResolvingCost(vr,regbank)
 4:   CoaleseBenefit   = CoalescingBenefit(vr,regbank)
 5:   return SpillCost + ConfResolvCost - CoaleseBenefit
 7: procedure ConflictResolvingCost(vr,regbank)
 8:   cost = 0
 9:   for all edges incident to vr in RCG do
10:     vr1 = the other end vertex of the edge
11:     if vr1's register bank is regbank then
12:       cost += the number of instructions
                referring both vr and vr1 as source operand
13:     end if
14:   end for
15:   return cost
16: procedure SpillCost(vr,regbank)
17:   NumOfInterferences = the number of vr's interfering live ranges
          whose register bank is regbank
          and priority higher than vr and has not been marked spill
18:   if NumOfInterferences >= NumOfAllocatableRegister then
19:     if CountOfSpills > local memory spill threshold then
20:       return vr's spill/reload count * SRAM Latency
21:     else return vr's spill/reload count * Local Memory Latency
22:     end if
23:   else return 0
24:   end if
```

Fig. 4. Cost Estimation

remove the copy instructions. To indicate this possibility, we add preference to each RCG node. When we see a copy instruction like `a = b`, we add `a` to `b`'s preference and vice versa. The CoalescingBenefit calculates the preference cost. It is essentially a product of a given weight and the number of elements that has been assigned a register bank in this set.

3.2 Live Range Splitting

Live range splitting is traditionally performed when failing to allocate a register to a live range, through inserting stores and reloads. However, in our framework, we prefer to do splitting at an earlier stage when we found that assigning the live range to any bank incurs a high cost. Instead of load/store, we use copy instruction to implement splitting and force the partitioned live ranges to different banks[11]. Compared to traditional splitting, this may result in additional copy instructions. However, it can further balance the register pressures between the two banks and reduce the number of loads/stores, which are much more expensive than copies.

To split the live range of a virtual register, we first build an *"induced graph"* of the region of the control flow graph (CFG) in which the virtual register is live. We check each connected component [12] of the subgraph to see if it can be allocated a register through comparing the number of available registers and the number of live ranges interfered with it. If a component does not seem to be able to get a register, we compute a min-cut for it using the method described in [13]. We add the cut edges to *CutEdgeSet* and insert compensation copies on

```
1: procedure SPLITTING(vr)
2:   build the induced graph for vr
3:   CutEdgeSet = NULL
4:   UPDATECOMPONENTINFO
5:   while not all component allocatable do
6:     for those components that are not allocatable do
7:       perform min-cut operation on this component
8:       add the cross edges to CutEdgeSet
9:       delete the cross edges from "induced graph"
10:    end for
11:    UPDATECOMPONENTINFO
12:  end while
13:  Assign each component a register bank
           based on the cost-benefit analysis
14:  Insert copy operations according to CutEdgeSet
```

Fig. 5. Live Range Splitting

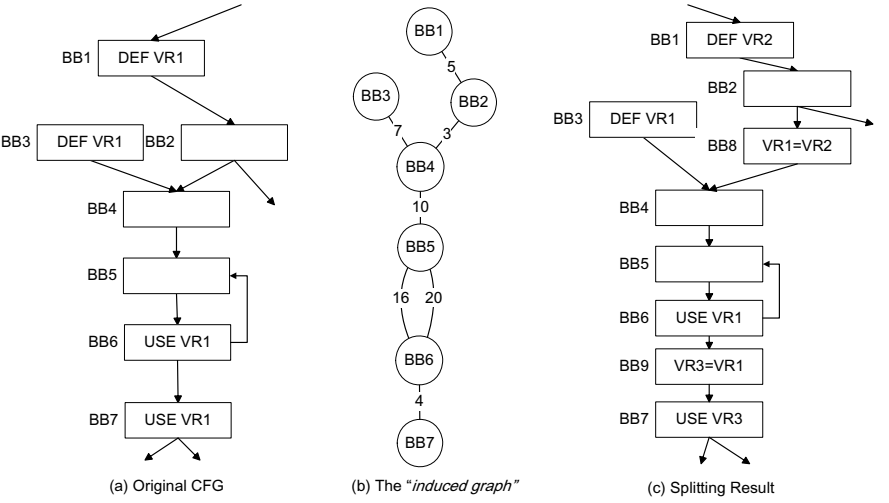


Fig. 6. An Example of Live Range Splitting

these edges later. This process iterated until all components becomes allocatable. After that, we rename each component with a new symbolic register, assign it to a register bank, and insert the corresponding copy operations based on the *CutEdgeSet*. This algorithm is show in Fig. 5.

Figure 6 shows an example of live range splitting. Fig. 6(a) shows the region of control flow graph in which $VR1$ is live. Fig. 6(b) shows the "induced graph" for $VR1$. The numbers on the edges are the frequencies of the control flowing through the edges. These numbers are obtained through profiling. The induced subgraph is connected, so we get the following partition:

$$P1 : \{BB1, BB2\},\ P2 : \{BB3, BB4, BB5, BB6, BB7\}$$

The *CutEdgeSet* contains $BB2 \rightarrow BB4$. Then, we apply the while loop again to these two partitions. Partition $P1$ is allocatable, while partition $P2$ is not. So we further cut it into two partitions:

$$P3 : \{BB3, BB4, BB5, BB6\},\ P4 : \{BB7\}$$

and the *CutEdgeSet* now changes to $BB2 \rightarrow BB4, BB6 \rightarrow BB7$. The third iteration of the while loop gets that all partitions are allocatable. The result is shown in Fig. 6(c).

4 Experimental Results

We evaluated our approach using three typical network applications written in `Baker` [8]:

L3-switch: performs L2 bridging or L3 forwarding of IP packets, depends on whether the source and destination of packet locates in a same virtual LAN.

Multi-protocol Label Switching (MPLS): routes packets on labels instead of destination IPs. This simplifies the processing of the packets and facilitates high-level traffic management. MPLS shares a large portion of code with L3-Switch.

Firewall: performs ordered rule-based classification to filter out unwanted packets. This application first assign flow IDs to packets according to user-specified rules and then drop packets for specified flow IDs. The flow IDs are stored in a hash table.

Table 1 shows some statistics of the benchmark applications. The data are gathered with a complete set of scalar optimizations and domain specific optimizations turned on [8]. The second column shows the lines of code information of a `Baker` implementation of these applications while the third column shows the number of instructions before bank conflict resolving. Only the instructions on the hot path and will be executed on ME are counted here. Column 4 gives the total number of GPR type virtual registers while column 5 shows the number of bank conflict.

We compared our approach with Zhuang's pre-RA bank conflict resolving method [9]. Table 2 shows the number of copy instructions and spills generated in the three benchmarks. As can be seen, Zhuang's method performs better in bank conflict resolving. After checking the RCGs, we find that most of the

Table 1. Benchmark Application Status

Application	LOC	# of Instr	# of VR	# bank conflicts
L3-Switch	3126	1378	806	238
MPLS	4331	1532	839	183
Firewall	2874	566	332	76

Table 2. Copy and Spill Status

	# of Copy Instrs		# Spill Operations	
	Pre-RA	Our	Pre-RA	Our
L3-Switch	0	11	35	25
MPLS	0	4	24	13
Firewall	0	1	34	16

Table 3. Distribution of Register Pressure Difference

Register Pressure	L3-Switch		MPLS		Firewall	
Difference	Pre-RA	Our	Pre-RA	Our	Pre-RA	Our
0	4.35%	5.80%	8.13%	16.25%	6.67%	11.67%
1	4.35%	21.73%	5.00%	32.50%	3.33%	35.00%
2	10.14%	34.78%	6.25%	15.00%	5.00%	28.33%
3	15.94%	16.67%	30.63%	25.00%	10.00%	16.67%
4	19.57%	10.14%	13.13%	6.88%	3.33%	8.33%
5	13.04%	6.52%	7.50%	2.50%	6.67%	0.00%
6	10.87%	2.90%	10.00%	0.63%	3.33%	0.00%
> 6	21.74%	1.45%	19.38%	1.25%	61.67%	0.00%
Weighted Mean	4.93	2.43	3.96	1.90	6.82	1.75

RCGs have only one or two nodes. Those RCGs with more than two nodes are essentially trees that do not have any cycles. On the other hand, our method outperforms Zhuang's method in that we generate fewer spills, which can be much slower than the copy instructions.

Table 3 shows the detailed distribution of the difference of register pressure between the two banks. The register pressure of a basic block is measured by the number of live ranges that live across that basic block. The data show the percentages of BBs with different register pressures. The last row shows the weighted mean of the register pressure difference between GPR bank A and GPR bank B. As can been seen, our approach can better balance the register pressure between the two banks.

5 Related Work

Banked register architecture has been used in VLIW processors to reduce the cycle time. [14][15]studied the bank assignment problem on such architectures based on the register component graph. The register component graph is a graph whose nodes are symbolic registers and arcs are annotated with the "affinity" that two registers have to be placed in the same register bank. After the register component graph being built, the problem becomes finding a min-cut of the graph so that the cost of the inter-bank copy is minimized. The bank constraints

in these architectures are different from that of IXP in 1) they do not have the *"two source-operand selection rule"*; 2) the inter-bank register copy instruction in these architectures is very expensive.

[16][17] discussed the memory bank conflict problem on some DSP processors. Many DSP processors, such as Analog Device ADSP2100, DSP Group PineDSPCore, Motorola DSP5600 and NEC uPD77016, etc. adopt banked memory organization. Such memory systems can fetch multiple data in a single cycle; given the data locate in different banks. Though compiler could optimize the allocation of the variables to avoid the delay caused by accessing a same bank in a single instruction, it's not mandatory.

Intel's MicroEngine C[18] is a C-like programming language designed for programming the IXP network processors in a relatively low level. It adds some extensions to C. One related to register bank assignment is the `__declspec` directive, which could be used to specify the allocation of the variables in the memory hierarchy. By default (without any `__declspec` qualifier), all the variables will be put in GPR. But this will increase the register pressure of GPR and cause spills in turn, which could be very expensive since MicroEngine C compiler would put them to SRAM. The programmers can do memory allocation manually using the `__declspec`. However, this puts too much burden on the programmer and is error-prone.

L. George, et al. [19] designed a new programming language named `Nova` for IXP 1200 network processor. They used integer linear programming to solve the bank conflict problem on IXP. While this method provides an upper bound on the performance benefit, the time complexity is too high to be practical.

X. T. Zhuang, et al. [9] discussed the register bank assignment problem for the IXP 1200 network processor. They proposed three approaches to solve the problem: performing bank assignment before register allocation, after register allocation, or at the same time in a combined way. They first build a register conflict graph (RCG) to represent the bank conflicts between symbolic registers. They showed that determining whether the virtual registers could be assigned banks without introducing copy instructions is equal to determining whether the RCG is bipartite. They proved the problem of making RCG bipartite with minimal cost is NP-complete by reducing the maximal bipartite sub-graph problem to it and suggested heuristic methods to solve the problem.

In [20], J. Park et al. presented a register allocation method for banked register file, in which only one register bank could be active at one time and registers are addressed using the register number in conjunction with bank number. No instructions except the inter-bank copy instruction can simultaneously access two banks. To solve this problem, they first divide the program into several allocation regions and then perform local register allocation using the secondary bank on these regions if they are deemed beneficial. Finally, the global register allocation would be performed on the primary bank and inter-bank copy operations would be inserted on the allocation region boundaries.

6 Conclusions

In this paper, we present a register allocation framework for banked register files with access constraints for the IXP network processors. Our approach relies on estimation of the costs and benefits of assigning a virtual register to a specific bank, as well as that of splitting it into multiple banks via copy instructions. We make the decision of bank assignment or live range splitting based on analysis of these costs and benefits. This helps to balance the register pressures among the banks. When splitting a live range, we use copy instructions instead of loads/stores and force the split live ranges to different banks. Though this may introduce additional copies, it can reduce the number of memory accesses significantly. Preliminary experiments show that compared with previous work, our framework can better balance the register pressure and reduce the number of spills, which in turn results in performance improvement.

References

1. Huang, J.H.: Network processor design. In: ASIC, 2003. Proceedings. 5th International Conference on. Volume Vol. 1 21-24. (2003) 26-33
2. Tseng, J.H., Asanović, K.: Banked multiported register files for high-frequency superscalar microprocessors. In: Proceedings of the 30th annual international symposium on Computer architecture. (2003)
3. Cruz, J.L., González, A., Valero, M., Topham, N.P.: Multiple-banked register file architectures. In: Proceedings of 27th annual international symposium on Computer Architecture. (2000) 316-325
4. Balasubramoniany, R., Dwarkadasy, S., Albonesi, D.H.: Reducing the complexity of the register file in dynamic superscalar processors. In: 34th International Symposium on Microarchitecture (MICRO-34). (2001) 237-248
5. Intel: Intel IXP2400 Network Processor Hardware Reference Manual. Intel Corporation. (2003)
6. Chow, F.C., Hennessy, J.L.: The priority-based coloring approach to register allocation. ACM Transactions on Programming Languages and Systems (TOPLAS) 12 (1990)
7. Intel: Intel IXP2400/IXP2800 Network Processor Programmers Reference Manual. Intel Corporation. (2003)
8. Chen, M.K., Li, X.F., Lian, R., Lin, J.H., Liu, L., Liu, T., Ju, R.: Shangri-la: achieving high performance from compiled network applications while enabling ease of programming. In: PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation, New York, NY, USA, ACM Press (2005) 224-236
9. Zhuang, X., Pande, S.: Resolving register bank con icts for a network processor. In: Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques(PACT' 03). (2003)
10. George, L., Appel, A.W.: Iterated register coalescing. In: POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages, New York, NY, USA, ACM Press (1996) 208-218
11. Farkas, K.I.: Memory-System Design Considerations For Dynamically-Scheduled Microprocessors. PhD thesis, University of Toronto (1997)

12. Sedgewick, R.: Algorithms in C++ Parts 5:Graph Algorithms. Addison Wesley/Pearson (2001)
13. Stoer, M., Wagner, F.: A simple min-cut algorithm. Journal of the ACM (JACM) (1997)
14. Hiser, J., Carr, S.: Global register partitioning. In: International Conference on Parallel Architectures and Compilation Techniques. (2000)
15. Jang, S., Carr, S., Sweany, P., Kuras, D.: A code generation framework for vliw architectures with partitioned register banks. In: Third International Conference on Massively Parallel Computing Systems. (1998)
16. Cho, J., Paek, Y., Whalley, D.: Efficient register and memory assignment of non-orthogonal architectures via graph coloring and mst algorithms. In: LCTES-SCOPES. (2002)
17. Keyngnaert, P., Demoen, B., de Sutter, B., de Bus, B., et al.: Con ict graph based allocation of static objects to memory banks. In: Semantics, Program Analysis, and Computing Environments for Memory Management. (2001)
18. Johnson, E.J., Kunze, A.R.: IXP 2400/2800 Programming: The Complete Microengine Coding Guide. Intel Press (2003)
19. George, L., Blume, M.: Taming the IXP network processor. In: PLDI. (2003)
20. Park, J., Lee, J.H., Moon, S.M.: Register allocation for banked register file. In: Language, Compiler and Tool Support for Embedded Systems LCTES. (2001) 39-47

Designing a Concurrent Hardware Garbage Collector for Small Embedded Systems

Flavius Gruian and Zoran Salcic

Department of Electrical and Computer Engineering,
The University of Auckland,
Private Bag 92019, Auckland, New Zealand
{f.gruian, z.salcic}@auckland.ac.nz

Abstract. Today more and more functionality is packed into all kinds of embedded systems, making high-level languages, such as Java, increasingly attractive as implementation languages. However, certain aspects, essential to high-level languages are much harder to address in a low performance, small embedded system than on a desktop computer. One of these aspects is memory management with garbage collection. This paper describes the design process behind a concurrent, garbage collector unit (GCU), a coprocessor to the Java Optimised Processor. The GCU, targeting small embedded real-time applications, implements a mark-compact algorithm, extended with concurrency support, and tuned for improved performance.

1 Introduction

Java, as a development language and run-time solution, seems to become increasingly attractive recently, even for embedded systems, given the plethora of Java-powered embedded processors [1, 2, 3, 4]. Nevertheless, few of these embedded platforms offer a true Java environment, including memory management with garbage collection. When present, garbage collection is in principle a software, stop-the-world approach, leading to poor performance systems. Although for high performance and desktop systems, both real-time and hardware supported garbage collection have been addressed by various research groups, there are few results for embedded systems with limited resources. In this paper, a concurrent garbage collection unit for the Java Optimised Processor (JOP, [5]) is described.

The paper is organised as follows. Section 2 mentions some of the relevant related work. The design methodology including goals and the design steps is given in Sect. 3. The used garbage collection algorithm is briefly described in Sect. 4, followed by the choices that remained unchanged throughout the design process in Sect. 5. The actual design iterations are detailed in Sect. 6. Section 7 discusses the implications of our solution, while Sect. 8 gathers our conclusions.

T. Srikanthan et al. (Eds.): ACSAC 2005, LNCS 3740, pp. 281–294, 2005.

2 Related Work

Improving the performance of garbage collectors by using parallelism or concurrency came under the attention of researchers long before Java was born [6, 7]. In the context of garbage collection, we use *parallelism* to describe collection work done by several processors at the same time, while *concurrency* refers to running the application (mutator) at the same time with the collector.

Many of the concurrent GC algorithms have their roots in the famous *Baker's algorithm* [8], which is however unsuitable for embedded systems we are interested in, due to its high demands on the memory size. Non-copying concurrent garbage collection algorithms, with lower demands on the memory are described in [9] and [10]. We decided to implement an incremental version of a mark-compact algorithm (see [11]), which avoids the fragmentation issues that may arise in a non-copying algorithm.

Although hardware accelerated GC was used in early LISP and Smalltalk machines, one of the first to address it from a real-time perspective is [12]. That paper proposed a garbage collected memory module (GCMM), employing *Baker*'s semi-space algorithm. [13] proposes the Active Memory Processor, a hardware GC for real-time embedded devices. That approach uses hardware reference counting and a mark-sweep algorithm, requiring extra memory that scales with the heap. In contrast, our solution is independent of the heap size.

3 Design Methodology

3.1 Goals

The work described in this paper started from the need of implementing a garbage collector for a Java Optimised Processor (JOP) system. Our JOP version is a three stage pipeline, stack oriented architecture. The first fetches bytecodes from a method cache and translates them to addresses in the micro-program memory. The method cache is updated on invokes and returns from an external memory. The second stage fetches the right micro-instruction and executes micro-code branches. The third decodes micro-instructions, fetches operands from the internal stack, and executes operations. Due to its organisation, JOP can execute certain Java bytecodes as single micro-instructions, while more the complex ones as sequences of micro-instructions or Java methods.

The goal evolved from implementing a software solution towards designing a hardware garbage collection unit, that can operate concurrently with the application. In addition, we wanted to achieve a modular and scalable solution, independent of the main memory (heap) size. Overall, we wanted an architecture suitable for resource-limited embedded, and possibly real-time, systems.

3.2 Approach

Adding a hardware garbage collection unit into an existing Java system that had none at all initially, involves a number of changes and additions that have

to fall into place. The class run-time images have to be augmented with GC information, which means modifying the application image generator. Certain bytecodes need to be modified to use the GC structures. The GC algorithm, the hardware GC unit that implements it, and concurrency support have to be all correct. Errors would be hard to identify and debug in such a system. Therefore we adopted a step-by-step approach for developing the GC unit. We started by implementing a pure software mark-compact, stop-the-world GC algorithm (see Sect. 6.1). In the next step, we wrote the hardware GC unit, simulated, synthesised, and tested it in an artificial environment, were we could generate and control the memory contents at word level (see Sect. 6.2). In the final step, we optimised the GCU and customised the JOP core in order to integrate the hardware GC solution with the Java platform in an efficient way (see Sect. 6.3). The first phase was thus dedicated to check the correctness of the class level GC information. The second phase focused on designing, implementing and partially evaluating the GC unit. The final phase addressed the integration of the GC unit with the JOP-based system, involving changes both in the processor and GCU, as well as in their communication mechanisms.

4 The GC Algorithm

At the base of our implementation resides a typical mark-compact algorithm (see [11] for a brief description). Our software implementation uses a *stop-the-world* collector, which preempts the mutator when a *new* requests more memory than the available contiguous free space. For the hardware unit, we adopted an incremental version of the aforementioned algorithm (see [11] for details on incremental GC). Using the *tricolor marking* abstraction [7], in the marking phase, grey objects are maintained into a specific *GC stack*. Every write access to an object will cause the mutator to push its handle onto the GC stack, as it might have been altered and needs to be re-scanned. Concurrently, the collector extracts grey objects (handles) from the stack, marks them, scans them for references and pushes all the unmarked handles it encounters into the stack. The marking phase finishes when the stack becomes empty.

In the compacting phase, the heap is scanned with two pointers, *ScanPtr* is used to examine successively all the objects in the heap, while *CopyPtr*, trailing behind, identifies the end of already compacted objects. As soon as a marked object is found by the *ScanPtr*, it is copied at the *CopyPtr*, its handle updated, and both *ScanPtr* and *CopyPtr* are advanced. If non-marked objects are encountered, their handle is recycled and only *ScanPtr* is advanced.

In this phase, the interaction between the mutator and collector becomes a bit more complex. The situations that we want to avoid by proper synchronisation are those in which either a write access is made on a partially copied object or a read access is made on an object about to be overwritten. The first situation appears because the write access might modify an already copied word, that would need to be recopied to maintain coherency. For the second situation consider the following scenario. The mutator translates a handle to

an instance pointer, meanwhile the collector moves the object and updates the instance pointer, and furthermore moves more objects until it overwrites the old copy of the first object. At this point the mutator holds a pointer to an invalid location. To avoid such situations we introduced read/write barriers in the form of object-level locks, as follows.

Whenever the mutator performs a read access on a certain object, it must set a read lock on that object's address, and reset the lock (unlock) when the access is completed. If the collector is about to overwrite the read-locked address, it stalls until the lock is reset. The write lock mechanism is rather similar. Whenever the mutator intends to do a write on a certain object, it must set a write lock on that object's handle, and unlock it when the access is completed. If the collector is copying or about to copy the object in question, it stalls its execution until the lock is reset. At this point, however, all the progress is reset, and copying resumes from the beginning of that object. Note that in principle we could have chosen instead to make the mutator wait for the object to be completely copied, if the write access occurs as the object is being copied. However, in that case the interference with the mutator would have been too significant, and decided to rather have the collector do more work than having the mutator wait. The drawback of this solution is though the fact that GC progress is not always guaranteed, as detailed in Sect. 7.

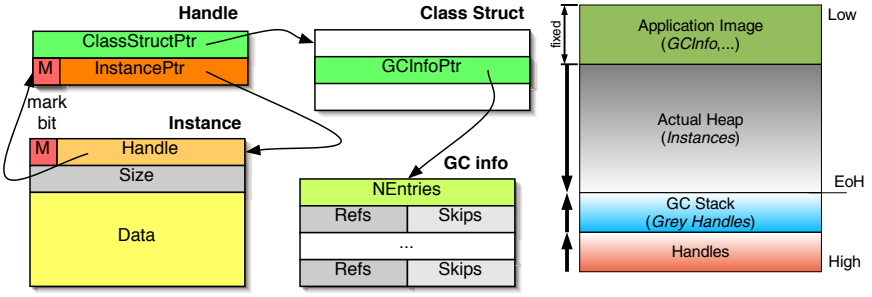
It is also important to notice that locking and unlocking are intended to occur at bytecode level, such that mutator threads cannot be preempted by other mutator threads while holding a lock. In fact this can be easily achieved in a JOP-based system, as it requires modifying the bytecodes for object accesses to also set and reset locks. This also means that only one lock can be set at any one time, observation that simplified the hardware architecture of the GCU.

5 Implementation Invariants

5.1 Data Structures

In a compacting GC algorithm, every time an object moves, all the pointers to that object would need to be updated. To overcome this we use *handles*, instance identifiers that are unique and constant throughout the object lifetime (see Fig. 1(a)). All accesses must first read the content of the object handle to find out the actual location of an instance.

Each instance header includes the size, the associated handle, and a mark bit. As instances are aligned to words, two extra bits are available in each pointer to an object – and we use one of them as a *marked* flag. Each handle is a pair of pointers, back to the instance and to the class structure, containing the necessary garbage collector information, *GCInfo*. The GCInfo structure is an array of pairs of half word values, the first containing the number of consecutive words that are references, and the second holding the number of consecutive words that are not references in the instance. A similar method of encoding information about the location of the references is used in [14]. The unused handles are maintained as a linked list, where the instance pointer is in fact the next free handle. Handles



(a) Object structure (b) Address space

Fig. 1. Implementation choices

recycled through GC are added at the tail of the list, while handles for new instances are acquired from the head of the list.

5.2 Address Space Organisation

The memory in the GC-enabled system is organised basically in two parts, the application image and the heap (see Fig. 1(b)). The application image contains the class information, methods bytecode, static members, and constants, including *GCInfo* (see Fig. 1(a)). The heap is divided in three contiguous regions. The first contains the instances, and is in fact the actual heap. The second, situated at the end of the address space, holds the object handles. Whenever the application runs out of handles, meaning that the free handles list becomes empty, a new batch of handles is built toward lower addresses, shrinking the actual heap space. This space is never returned to the heap, but will be used as handles for the rest of the application life time. Finally, the third region, the *GCStack*, is only temporary. This extends between the handle space and grows downwards over the actual heap when the GC algorithm is in its marking phase. *GCStack* is in fact a handle stack used by the breadth-first traversal and marking of the live objects (see Sect. 4).

6 Design Iterations

Going from a system without any GC support to a concurrent hardware GC involves a significant number of additions and modifications. The GC information for each class has to be added in the application image, which means modifying the image generator. Locking mechanisms have to be added for the bytecodes accessing objects, which for JOP meant changes in the microcode. The GC unit had to be built and tested properly, if possible on a rather realistic setup, before adding it into the JOP-based system. Finally, the system needed to be optimised and adapted together with the GCU, which involves changes both in the GCU,

the main processor (JOP), and communication structure. All these steps were gradually taken, in order to detect and tackle problems more efficiently.

6.1 Software, Stop-The-World on the Target Platform

At first, we implemented the GC in software on a system containing a custom version of the Java Optimised Processor (JOP, [5]). In this version, a GC cycle is performed in a stop-the-world manner, whenever a *new* cannot be performed because of the lack of free memory. Whenever this happens, the following steps are taken. The JOP stack is scanned for handles (root references), which are pushed into the GC stack. The MARK phase starts at this point. Live objects are traversed in a breadth-first manner, using the GC stack to store detected but not scanned handles. Once the GC stack is emptied, the COMPACT phase starts. Marked objects are compacted at the beginning of the heap and the marked flag cleared. The cycle terminates once all the heap objects are scanned. At this point *new* resumes its normal operation, by allocating the required heap space. Handles are also managed inside the *new* operation. Besides writing the actual GC code, a number of issues must be addressed.

Identifying Handles. Initially, faithful to the JVM specification, JOP could not distinguish between handles and values on the stack. However, this is a must in order to register root references. The problem of identifying references is addressed for example in [15]. One solution is to use two separate stacks, one for references and one for values [14]. Another solution would be to tag each stack word with an extra bit, signifying reference. However, to avoid altering the JOP data path and the micro-instruction, we adopted the following, conservative method. As the handles are usually stored in a dedicated area, one can assume that any stack word is a reference if and only if points inside the handle area.

Bytecode Modifications. Bytecodes in JOP are implemented either as micro-programs and/or as Java functions. To support our garbage collection solution, bytecodes accessing objects were modified to translate handles into references.

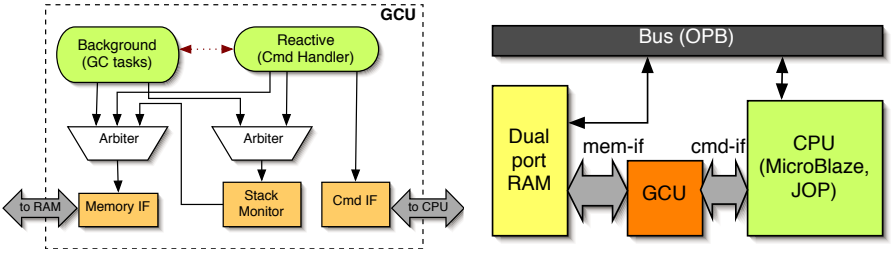
Application Image Impact. Finally, the application image was extended with GC functionality (two classes) and augmented with the information necessary during the garbage collection cycle (GC info in Fig. 1(a)). The images increase in size by about 350 words (14%) including the GC required class structures and methods. This increase is marginally dependent on the number of objects, as each class is extended with 3-5 words of GC information. Certainly, the application image generator can be further improved to reduce these images even more.

Experimental Evaluation. All of the systems used for evaluation in this paper were synthesised and run at 25MHz on a Xilinx Spartan2e (XC2S600e) FPGA. At this point we were mainly interested in the correctness of the GC implementation rather than in its performance. Nevertheless, measurements on similar benchmarks as the ones used later on in Sect. 6.2 revealed that a GC cycle takes in the range of tens of milliseconds, which is expected, considering the low CPU performance. This is yet another reason for exploring a hardware GC.

6.2 Hardware, Concurrent on a Test Platform

A concurrent collector should be able to both carry out the garbage collection and handle commands from the application at the same time. Let us name these tasks the *Background*, performing normal GC operations and the *Reactive*, handling commands. The *Reactive* would be required at points to access the GC stack, in order to store root pointers or grey references. The *Background* would also need to access the GC stack to push and pop handles as it marks live objects. Furthermore, stack operations should be atomic. It becomes apparent that a common resource, a *Stack Monitor*, used by both processes, is the solution. Additionally, the two processes and the stack monitor would in fact access the system memory. The architecture we decided to implement is depicted in Fig. 2(a). To make the design easier to port, we decided to use a single, common, memory interface (*Memory IF*) for all the accesses to the system memory. This can be easily rewritten to support all kinds of memory access protocols. Similarly, the interface through which GCU receives commands (*Cmd IF*) can be adapted to support various kinds of processor interfaces. All accesses to common resources are handled by arbiters using fixed priorities. The *Reactive* process has the highest priority, stack operations have medium priority, while the *Background* process has the lowest priority. In fact all modules from Fig. 2(a), except the arbiters, are synchronous (FSMs) exchanging values through hand-shaking. Most of the time the *Reactive* and *Background* processes synchronise and communicate indirectly through stack and memory accesses. However, for some commands (such as read/write lock, unlock, and new) the two processes communicate directly through hand-shaking, as the *Reactive* process must acknowledge the execution of these commands to the main processor. The modularity of the design, although very easily adaptable to various memory systems and processors, is also its downfall, as detailed in Sect. 6.3.

System Overview. The placement of the GCU inside the complete system was chosen to minimise the interference with the CPU, taking advantage of the multi-port memory normally available on FPGAs. The generic solution is depicted in Fig. 2(b). GCU is directly connected to the CPU (*cmd-if*), and to



(a) Internal architecture (b) Placement in the system

Fig. 2. Hardware garbage collection unit

a second port of the memory architecture (*mem-if*). This way of integrating the GCU into the system will remain unchanged throughout our design process. Furthermore, most components around the GCU also remain fixed, as our CPU uses the On-chip Peripheral Bus (OPB) as system bus.

In order to evaluate our solution in a realistic but highly controllable environment, we connected the GCU into a MicroBlaze-based system. As command interface we used a dual Fast Simplex Link (duplex FSL), while the memory interface connects directly through a Block RAM port. Thus, we had a finer grain control over the memory structures used by the GC, as we used C written code to emulate, construct and verify object images. We could also use tested components and interfaces around the GCU, taking advantage of the better development tool support for MicroBlaze. Finally, we also wanted to achieve a rather processor independent design, portable to other systems.

Runtime Perspective. The commands implemented initially by the GCU are gathered in Table 1. From the programmer's point of view, the concurrency-related commands are *rdlock*, *wrlock*, *unlock*, and *waitidle*. The first three are used to ensure the correct access to objects, while the last is used to join the GC process with the main application. Some of these commands are synchronous, causing the GCU to send an acknowledgement (*GCU Ack* column), while others can be just issued without expecting a reply from the GCU.

A GC cycle is triggered by the CPU, by issuing a *stackinit* command to the GCU (see Fig. 3.a), which goes from IDLE to MARK phases. The CPU registers root pointers by pushing them in the GC stack via *rootref* commands. Concurrently, as soon as the GC stack contains handles, the GCU starts marking objects. Once all the root references have been pushed into the GC stack, the CPU issues a *docmpct* command, allowing the GCU to start the COMPACT phase. Next, the CPU can start executing application code, using *rdlock*, *wrlock*, *unlock*, and *new* as in Fig. 3.b. As soon as all the live objects have been marked (the GC stack becomes empty), the GCU begins the COMPACT phase. The CPU can wait for the GCU to finish by issuing a *waitidle*, update the end of heap received from the GCU, and maybe start a new GC cycle.

Inside the application code, every time a new object is created, the CPU must issue a *new* to the GCU and wait for an acknowledgement (see Fig. 3.b). Note that acquiring a new handle and determining the object size is the respon-

Table 1. Initial GCU Commands

Command	Words	GCU Action	GCU Ack
[rd,wr]lock h	1	Locks object with handle *h*.	obj address
unlock	1	Unlocks a previously locked object.	yes, any
waitidle	1	Waits until GCU is IDLE.	end of heap
stackinit a	2	Sets GC stack base at *a*. Starts MARK phase.	none
rootref h	2	Registers *h* as root reference.	none
docmpct h	2	Starts COMPACT. Freed handles appended to *h*.	none
new h, s	2	Creates an object of size *s* and handle *h*.	end of heap

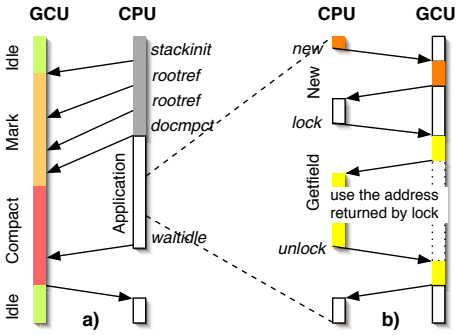


Fig. 3. GCU-CPU synchronisation: a) A GC cycle, b) New and the lock mechanism

sibility of the software *new object* function or micro-code. Accesses to objects data (GET/PUTFIELD, *ALOAD, *ASTORE, ...) must be preceded by *rd/wrlock* on the object handle. The CPU must wait for an acknowledgement of the lock from the GCU before translating the handle into a real address and accessing the object. The CPU must call *unlock* as soon as the access is completed.

Experimental Evaluation. We initially compared the performance our hardware approach to a software version of the same algorithm, both used in a stop-the-world manner. The software version was coded in C and compiled with *gcc* using the highest optimisation level. The mutator was a simple application that creates alternatively elements from two linked lists, and keeping both, one or neither of the lists alive followed by a GC cycle. The three situations reflect three different memory configurations, when no objects are moved, half of the objects are moved and finally when all objects are discarded. The GC cycles were timed using an OPB_TIMER core. The results for lists of ten, thirty, and fifty elements are depicted in Fig. 4. Note that the GCU performs consistently around four times faster than the software version. Some of the speed-up is due to using a memory port directly instead of the OPB, as in the software version. However, the GCU used in this experiment is not optimised as a stop-the-world version, but instead implements all the features required for concurrent GC.

Looking at the overhead introduced by synchronisation into the object access latency, one lock/unlock pair takes about 19 clock cycles altogether. This is a much larger overhead than we initially intended. The reason behind this large overhead lay in the architecture of the GCU which was constrained by the capabilities of the MicroBlaze core. In particular, for the lock-related commands, they need to propagate through the FSL, are decoded in the *Reactive* process, and finally reach the *Background* process. Furthermore, an acknowledgement follows the same lengthy path back to the processor. Using a dedicated channel only for the locked address or handle, directly from the processor to the GCU, the latency would reduce considerably. This would imply modifying the proces-

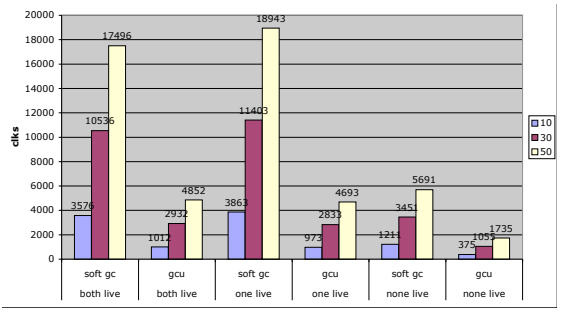


Fig. 4. Stop-the-world GC latency of the GCU vs. a software, C version (soft-gc) for an application using two linked lists with 10, 30, and 50 elements alternatively created

sor core, which is impossible for proprietary IP such as MicroBlaze. However, for the GCU version intended for JOP (an open core) this is not a problem.

6.3 Improved Hardware, Target Platform Specific

Integrating the GCU with the target platform, centred around JOP, imposed and at the same time allowed for a number of changes in the CGU command and memory interface. In this iteration the system used a standard Local Memory Bus (LMB) for memory interface, and a custom command interface (called Fast Command Access – FCA), while the rest of the architecture remained basically the same (see Fig. 2(b)). Micro-architectural changes in JOP and also GCU had to be implemented in order to employ the specialised FCA. Compared to the pure software approach, there has been migration of the software functionality into hardware, which affected both the application image and the device utilisation.

Fast Command Access Interface. The FCA was designed as an alternative to the FSL, in order to reduce the lock and unlock overhead. The JOP core was extended as follows. First, a *fast command register* (FCR) was introduced inside the processor, register directly seen by the GCU (*CmdHi* in Fig. 5). This register holds the currently locked handle, value that needs to be persistent while the lock is in effect. To allow extended commands (double word) to reach the

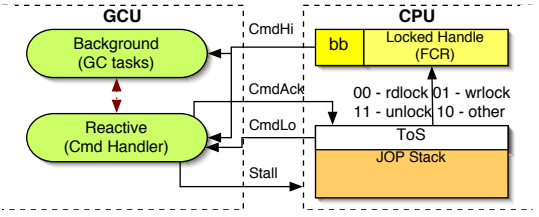


Fig. 5. Fast Command Access interface: a deeper integration of the GCU and CPU

Table 2. GCU, JOP, and systems device utilisation (on a Xilinx Spartan2e, XC2S600e)

Unit resources	GCU only	JOP only	**Full system resources**	JOP, RAM, IPs	JOP, GCU RAM, IPs
Slice FF	900	400	Slices	1543 (22%)	3053 (44%)
4LUT	2966	1783	BRAMs	71 (98%)	71 (98%)

GCU, the top of the stack from JOP is also made visible to the GCU (*CmdLo* in Fig. 5). The GCU can also send back to JOP words through a *CmdAck* signal. Furthermore, the GCU can stall JOP, if needed, through a *Stall* line. The JOP micro-instruction set was extended with two new ones: **stfc** pops the stack and stores the value into the FRC, and **ldfc** pushes *CmdAck* into the stack.

The GCU uses the FCA interface as follows. Read, write locks, and unlocks require one word (*CmdHi*) for identification. The *Background* process can readily use handles stored in *CmdHi* to determine lock situations. At the same time, the *Reactive* process decodes both *Cmd* lines and drives the *CmdAck* when necessary. In particular, read/write lock latency is reduced to one memory access, needed by the *Reactive* process to translate the object handle into an address.

Application Image Impact. The software GC functionality is now obsolete, being implemented by the GCU. The application image has been reduced by approximately 100 words compared to the pure software implementation.

Device Utilisation Impact. Adding the GCU doubles the area used for the JOP system (see Table 2), which includes along with JOP itself an OPB bus connecting a Block RAM, a UART, a timer, and some general purpose I/O cores. The synthesis tool reports the GCU clock frequency at 78MHz on the XC2S600e.

Experimental Evaluation. Using a JOP-based system, we evaluated the performance of the GCU as a stop-the-world garbage collector, with the same application from Sect. 6.2. As expected, the figures were similar to the ones reported for the MicroBlaze test system (see Fig. 4, GCU), as only the interface to the GCU changed. Compared to the pure Java GC, the speed-up achieved by using the GCU is impressive, a GC cycle being almost a hundred times shorter.

Next, using ModelSim, we examined the overhead introduced in JOP by the synchronisation required for some of the GCU commands, namely *(rd/wr)lock*, *unlock*, and *new*. The locking mechanism is employed to make sure that an object does not move during an access, while *new* is needed for allocating new objects. The overhead varies between 2 and 7 clock cycles for lock related operations and 15 clock cycles for *new*. Note that these figures reveal a lower overhead for locking and unlocking objects compared to the MicroBlaze-tested solution (Section 6.2), due to its improved GCU-CPU communication. However, it makes more sense to look at this overhead in the context of bytecodes, as all those using references need to employ the locking mechanism. In particular, the micro-program associated with bytecodes reading(writing) object contents needs to be extended with read (write) locks on the object handle before the access and conclude with

Table 3. Maximal synchronisation overhead in clock cycles, per bytecode class

read access bytecodes				write access bytecodes			
class	latency (clock cycles)			class	latency (clock cycles)		
	before	after	change		before	after	change
GEFIELD	28	31	11%	PUTFIELD	30	45	50%
*ALOAD	41	44	7%	*ASTORE	45	60	33%
ARRAYLENGTH	15	18	20%	NEW,	Java		
INVOKE*	> 100	+3	< 3%	*NEWARRAY	methods		< 1%

an unlock. As each bytecode is implemented as a sequence of micro-instructions, the actual overhead of the locking mechanism is even smaller at this level. For bytecodes implemented as Java methods by JOP (NEW, NEWARRAY, ANEWARRAY), the overhead becomes negligible (see Table 3). The most strongly affected is PUTFIELD, that increases its execution time by 50%. Nevertheless, the impact these have on the applications overall, depends highly on the specific application.

7 Discussion

Ensuring GC Progress. One of the goals we set for our GC solution is very low interference with the application, which translates into low latency for accessing objects. In other words, *lock/unlock* and *new* must be as fast as possible. Furthermore, deterministic times for these operations are also desired for real-time applications. Although these goals have been achieved, there is a price to pay on the GC side. In particular, write locks/unlocks in the middle of an object move force a rollback of the progress to the beginning of that object. For large objects, frequently written, it could happen that the GC will never be able to finish moving the object in question. This can happen if the time between two write accesses of an object is shorter than the time needed to move the whole object. Nevertheless, there are several ways of ensuring progress in such cases. Offline analysis can reveal the maximum size of an object for which the GC still makes progress. One can then either rewrite the application code or split the objects into smaller objects. Another possibility would be a time-out behaviour, delaying the application when the GC makes no progress for a certain time interval.

Another Processor as GCU. For more flexibility, one can imagine using a second processor instead of a hardware GCU. The initial tendency is to implement the *Reactive* process (see Sect. 6.2) as an interrupt handler, leaving the *Background* process as the normal operation mode. However, the interrupt handling latency for a general purpose processor is at least in the range of tens of clock cycles, making the lock/unlock latency too large to be useful. Nevertheless, with the advent of reactive processors, such as ReMIC [16], the possibility of a processor-based GCU appears feasible and exciting.

Real-Time Considerations. Standard garbage collectors are allocation triggered, meaning that whenever the free memory decreases under a certain limit, a GC cycle is started. The GCU we presented in this paper may very well be employed in that manner. However, our GCU is more suited for a *time-triggered garbage collection* (TTGC) approach, offering better real-time performance [17]. As GC in our approach is truly concurrent, it makes more sense to view a GC cycle as a task, rather than as small increments packed into allocation steps. This task is then treated no differently than the rest of the tasks in a real-time system, as long as it can be performed often enough to provide the necessary free memory. Theorem 1 in [17] provides an upper bound for the GC cycle time that guarantees that there will always be enough memory for allocation. In our case, this is the period our GCU must be initialised and allowed to run a full GC cycle. However, having the CPU and GCU synchronise now and then via locks introduces additional delays.

Another way to employ the GCU would be to run it constantly, staring a new cycle as soon as the current one finishes. However, depending on the application, this may lead to a large amount of wasted work and energy.

8 Conclusion

The current paper presented a hardware garbage collection unit, designed to work concurrently with the CPU in Java-based embedded system. Given the complexity of adding a concurrent GC into a system without any GC support at all, a gradual design approach was taken, to identify and fix problems easier.

To satisfy the requirements of minimal interference GC, our solution involves not only an efficient GC unit, but also specialised support in the processor, necessary for fast CPU-GCU interaction. The dedicated hardware GCU itself consists of two processes, one dedicated for handling commands and synchronising with the CPU, and the other implementing a mark-compact GC algorithm.

As a stop-the-world garbage collector, our GCU is four times faster than a highly optimised C solution, and orders of magnitude faster than a Java solution. As a concurrent solution, the locking mechanism introduced to keep memory consistency introduces a small overhead in the system, bringing the benefit of running in parallel with the application. Finally, our solution seems to be suitable for real-time applications, when time-triggered garbage collection is employed.

References

1. Sun: PicoJava-II microarchitecture guide. Technical Report 960-1160-11, Sun Microsystems (1999)
2. Hardin, D.S.: aJile systems: Low-power direct-execution Java microprocessors for realtime and networked embedded applications. (aJile Systems Inc.)
3. : Moon2- 32 bit native Java technology-based processor. (Vulcan Machines Ltd.)
4. : Lightfoot 32-bit Java processor core. (Digital Communication Technologies)
5. Schoeberl, M.: JOP: A java optimized processor. In: Workshop on Java Technologies for Real-Time and Embedded Systems. (2003)

6. Steele, G.L.: Multiprocessing compactifying garbage collection. Communications of the ACM **18** (1975) 495–508
7. Dijkstra, E.W., Lamport, L., Martin, A.J., Scholten, C.S., Steffens, E.F.M.: On-the-fly garbage collection: An exercise in cooperation. Communications of the ACM **21** (1978) 966–975
8. Baker, H.G.: List processing in real-time on a serial computer. Communications of the ACM **21** (1978) 280–294
9. Boehm, H.J., Demers, A.J., Shenker, S.: Mostly parallel garbage collection. In: Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation. (1991) 157–164
10. Printezis, T., Detlefs, D.: A generational mostly-concurrent garbage collector. In: Proceedings of the ACM SIGPLAN International Symposium on Memory Management. (2000) 143–154
11. Wilson, P.R.: Uniprocessor garbage collection techniques. In: Proc. Int. Workshop on Memory Management, Springer-Verlag (1992)
12. Schmidt, W.J., Nilsen, K.D.: Performance of a hardware-assisted real-time garbage collector. In: *Proceedings of the Sixth Intl. Conf. on Architectural Support for Programming Languages and Operating Systems.* (1994) 76–85
13. Srisa-an, W., Lo, C.T.D., Chang, J.M.: Active memory processor: A hardware garbage collector for real-time java embedded devices. IEEE Transactions on Mobile Computing **2** (2003) 89–101
14. Ive, A.: Towards an embedded real-time Java virtual machine. Lic.Thesis 20, Dept. of Computer Science, Lund University (2003)
15. Agesen, O., Detlefs, D.: Finding references in java stacks. In: OOPSLA'97 Workshop on Garbage Collection and Memory Management. (1997)
16. Salcic, Z., Hui, D., Roop, P., Biglari-Abhari, M.: ReMic - design of a reactive embedded microprocessor core. In: Proceedings of Asia-South Pacific Design Automation Conference. (2005)
17. Gestegard-Robertz, S., Henriksson, R.: Time-triggered garbage collection. In: Proceedings of the ACM SIGPLAN Langauges, Compilers, and Tools for Embedded Systems. (2003)

Irregular Redistribution Scheduling by Partitioning Messages

Chang Wu Yu, Ching-Hsien Hsu, Kun-Ming Yu, C.-K. Liang, and Chun-I Chen

Department of Computer Science and Information Engineering,
Chung Hua University, Hsinchu, Taiwan 300, R.O.C
{cwyu, chh, yu, ckliang}@chu.edu.tw

Abstract. Dynamic data redistribution enhances data locality and improves algorithm performance for numerous scientific problems on distributed memory multi-computers systems. Previous results focus on reducing index computational cost, schedule computational cost, and message packing/unpacking cost. In irregular redistribution, however, messages with varying sizes are transmitted in the same communication step. Therefore, the largest sized messages in the same communication step dominate the data transfer time required for this communication step. This work presents an efficient algorithm to partition large messages into multiple small ones and schedules them by using the minimum number of steps without communication contention and, in doing so, reducing the overall redistribution time. When the number of processors or the maximum degree of the redistribution graph increases or the selected size of messages is medium, the proposed algorithm can significantly reduce the overall redistribution time to 52%.

1 Introduction

Parallel computing systems have been extensively adopted to resolve complex scientific problems efficiently. When processing various phases of applications, parallel systems normally exploit data distribution schemes to balance the system load and yield a better performance. Generally, data distributions are either regular or irregular. Regular data distribution typically employs BLOCK, CYCLIC, or BLOCK-CYCLIC(c) to specify array decomposition [14, 15]. Conversely, an irregular distribution specifies an unevenly array distribution based on user-defined functions. For instance, High Performance Fortran version 2 (HPF2) provides a generalized block distribution (GEN_BLOCK) [19, 20] format, allowing unequally sized messages (or data segments) of an array to be mapped onto processors. GEN_BLOCK paves the way for processors with varying computational abilities to handle appropriately sized data.

Array redistribution is crucial for system performance because a specific array distribution may be appropriate for the current phase, but incompatible for the subsequent one. Many parallel programming languages thus support run-time primitives for rearranging a program's array distribution. Therefore developing efficient algorithms for array redistribution is essential for designing distributed memory compilers for those languages. While array redistribution is performed at run time, a trade-off occurs between the efficiency of the new data rearrangement for the coming phase and the cost of array redistributing among processors.

T. Srikanthan et al. (Eds.): ACSAC 2005, LNCS 3740, pp. 295 – 309, 2005.

Performing data redistribution consists of four costs: index computational cost T_i, schedule computational cost T_s, message packing/unpacking cost T_p and data transfer cost. The data transfer cost for each communication step consists of start-up cost T_u and transmission cost T_t. Let the unit transmission time τ denote the cost of transferring a message of unit length. The total number of communication steps is denoted by C. Total redistribution time equals $T_i+T_s+\sum_{i=1}^{i=c}(T_p+T_u+m_i\tau)$, where m_i =Max{d_1, d_2, d_3, .., d_k} and d_j represents the size of message scheduled in i^{th} communication step for j=1 to k.

Previous results focus on reducing the former three costs (i.e., T_i, T_s, and T_u). In irregular redistribution, messages of varying sizes are scheduled in the same communication step. Therefore, the largest size of message in the same communication step dominates the data transfer time required for this communication step. Based on the fact, this work presents an efficient algorithm to partition large messages into multiple small ones and schedules them by using the minimum number of steps without communication contention and, in doing so, reducing the overall redistribution time. Specifically, the minimum value of T_s, and C are derived, along with the value of m_i reduced by shortening the required communication time for each communication step. When the number of processors or the maximum degree of the redistribution graph increases or the selected size of messages is medium, the proposed algorithm can significantly reduces the overall redistribution time to 52%. Moreover, the proposed algorithm can be applied to arbitrary data redistribution while slightly increasing the communication scheduling time.

The rest of the paper is organized as follows. Section 2 presents necessary definitions and notations. Next, Section 3 describes the basic graph model along with related work. The main contribution of the paper is shown in Section 4. We also conduct simulations in Section 5 to demonstrate the merits of our algorithm. Finally, Section 6 concludes the paper.

2 Definitions and Notations

A *graph* G consists of a finite nonempty vertex set together with an edge set. A *bipartite graph* G =(S, T, E) is a graph whose vertex set can be partitioned into two subsets S and T such that each of the edges has one end in S and the other end in T. A typical convention for drawing a bipartite graph G=(S, T, E) is to put the vertices of S on a line and the vertices of T on a separate parallel line and then represent edges by placing straight line segments between the vertices that determine them. In this convention, a drawing is *biplanar* if edges do not cross, and a graph G is *biplanar* if it has a biplanar drawing.

Let $N(v)$ denote the set of vertices which are adjacent to v in G. The ends of an edge are said to be *incident* with the edge. Two vertices which are incident with a common edge are *adjacent*. A *multi-graph* is a graph allowing more than one edge to join two vertices. The degree $d_G(v)$ of a vertex v in G is the number of edges of G incident with v. We denote the maximum degree of vertices of G by $\Delta(G)$.

A *complete bipartite graph* G =(S, T, E) is a graph such that each vertex of S is joined to each vertex of T; if $|S|=m$ and $|T|=n$, such a graph is denoted by $K_{m,n}$. An ordering

of S (T) has the *adjacency property* if for each vertex $v \in T(S)$, $N(v)$ contains consecutive vertices in this ordering. The graph $G=(S, T, E)$ is called a *doubly convex-bipartite graph* if there are orderings of S and T having the adjacency property [24].

The coloring is *proper* if no two adjacent edges have the same color. An edge with identical ends is called a *loop*. A *k-edge coloring* of a loopless graph G is an assignment of k colors to the edges of G. G is *k-edge colorable* if G has a proper k-edge coloring. The *edge chromatic number* $\chi'(G)$, of a loopless graph G, is the minimum k for which G is k-edge-colorable. A subset M of E is called a *matching* in $G=(V, E)$ if its elements are links and no two are adjacent in G. Note that the each edge set with the same color in a proper edge coloring forms a matching. At last, most graph definitions used in the paper can be found in [22].

3 Graph Model and Related Work

A bipartite graph model will be introduced to represent data redistributions first. Next, related work will be surveyed briefly.

3.1 Graph Model

Any data redistribution can be represented by a bipartite graph $G=(S, T, E)$, called a *redistribution graph*. Where S denotes source processor set, T denotes destination processor set, and each edge denotes a message required to be sent. For example, a Block-Cyclic(x) to Block-Cyclic(y) data redistribution from P processors to Q processors (denoted by BC (x, y, P, Q)) can be modeled by a bipartite graph $G_{BC(x, y, P, Q)}=(S, T, E)$ where $S=\{s_0, s_1, \ldots, s_{|s|-1}\}$ ($T=\{t_0, t_1, \ldots, t_{|t|-1}\}$) denotes the source processor set $\{p_0, p_1, \ldots, p_{|s|-1}\}$ (destination processor set$\{p_0, p_1, \ldots, p_{|t|-1}\}$) and we have $(s_i, t_j) \in E$ with weight w if source processor p_i has to send the amount of w data elements to destination processor p_j. For simplicity, we use BC (x, y, P) to denote BC (x, y, P, P). Figure 1 depicts the a data redistribution pattern $BC(1, 4, 4)$, and its corresponding redistribution graph $G_{BC(1, 4, 4)}$ is shown in Figure 2.

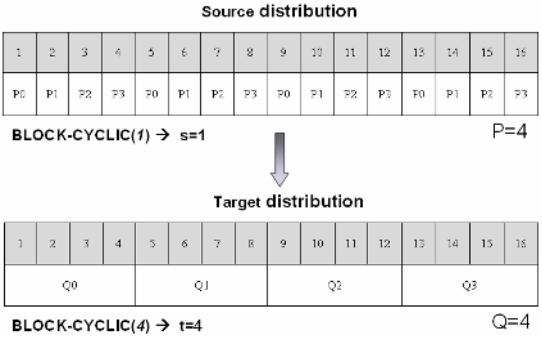


Fig. 1. A data redistribution pattern $BC(1, 4, 4)$

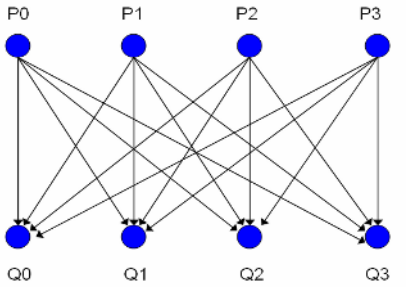


Fig. 2. The redistribution graph $G_{BC(1, 4, 4)}$ is a complete bipartite graph

Similarly, GEN_BLOCK data redistribution from P processors to Q processors (denoted by GB (P, Q)) can also be modeled by a bipartite graph $G_{GB(P, Q)}$=(S, T, E). For example, a GB(4, 4) with its redistribution graph $G_{GB(4, 4)}$ is depicted in Figure 3 and 4.

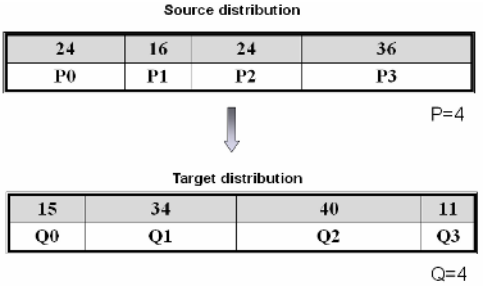


Fig. 3. GEN_BLOCK data redistribution GB(4, 4)

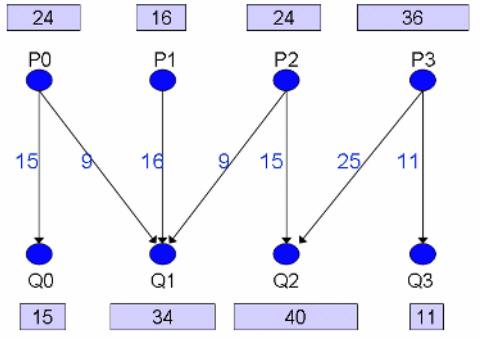


Fig. 4. A redistribution graph $G_{GB(4, 4)}$

Note that the the redistribution graphs of GEN_BLOCK are biplanar graphs, which are subgraph of doubly convex-bipartite graphs. Moreover, the next theorem shows interesting properties of biplanar graphs.

Theorem 1. The following four statements are equivalent [25-27]:

(1) A bipartite graph G is biplanar.
(2) The graph G is a collection of disjoint caterpillars.
(3) The graph G contains no cycle and no double claw.
(4) The graph G^* that is the remainder of G after deleting all vertices of degree one, is acyclic and contains no vertices of degree at least three.

Here a *caterpillar* is a connected graph that has a path called the *backbone b* such that all vertices of degree larger than one lie on b; and a *double claw* graph is depicted in Figure 5.

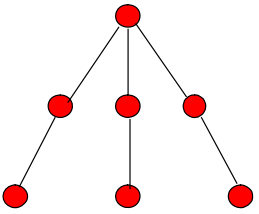


Fig. 5. A double claw

3.2 Related Work

Techniques for regular array redistribution can be classified into two groups: the communication sets identification and communication optimizations. The former includes the *PITFALLS* [17] and the *ScaLAPACK* [16] methods for index sets generation. Park *et al.* [14] devised algorithms for BLOCK-CYCLIC data redistribution between processor sets. Dongarra *et al*. [15] proposed algorithmic redistribution methods for BLOCK-CYCLIC decompositions. Zapata *et al*. [1] designed parallel sparse redistribution code for BLOCK-CYCLIC data redistribution based on *CRS* structure. Also, the *Generalized Basic-Cycle Calculation* method was presented in [3]. Techniques for communication optimizations provide different approaches to reduce the communication overheads in a redistribution operation. Examples are the processor mapping techniques [10, 12, 4] for minimizing data transmission overheads, the multiphase redistribution strategy [11] for reducing message startup cost, the communication scheduling approaches [2, 7, 13, 21] for avoiding node contention, and the strip mining approach [18] for overlapping communication and computational overheads.

With respect to irregular array redistribution, previous work focused on the indexing and message generation or the communication efficiency. Guo *et al.* [9] presented a symbolic analysis method for communication set generation and reduced the communication cost of irregular array redistribution. To reduce communication cost, Lee *et al.* [12] presented four logical processor reordering algorithms on irregular array redistribution. Guo *et al.* [19, 20] proposed a divide-and-conquer algorithm to perform irregular array redistribution. By using Neighbor Message Set (NMS), their algorithm divides communication messages of the same sender (or receiver) into groups; the resulting communication steps will be scheduled after merging those NMSs according

to the contention status. In [21], Yook and Park proposed a relocation algorithm consisting of two scheduling phases: the list scheduling phase and the relocation phase. The list scheduling phase sorts global messages and allocates them into communication steps in decreasing order. When a contention happened, the relocation phase performs a serial of re-schedule operations, which leads to high scheduling overheads and degrades the performance of a redistribution algorithm.

4 Irregular Redistribution Scheduling

In general, most data redistribution scheduling algorithms encounter a difficulty: shortening the overall communication time without increasing the number of communication steps at the same time. In this section, we devise an efficient scheduling algorithm to drastically reduce the total communication time with the minimum number of communication steps.

4.1 Motivation

Given a redistribution graph G with its edge coloring, the edges colored the same is a matching of G; thus represents a set of conflict-free data communication. Accordingly, for a given data redistribution problem, a conflict-free scheduling with the minimum number of communication steps can be obtained by coloring the edges of the corresponding redistribution graph G. When G is bipartite, it is well known that $\chi'(G)=\Delta(G)$ [22]. As a result, the minimum number of required communication steps equals the maximum degree Δ of the given distribution graph G.

Previous work is equivalent to finding out an edge colorings $\{E_1, E_2, E_3, \ldots, E_\Delta\}$ of G so that $\sum_{i=1}^{\Delta} \max \{w_k | e_k \in E_i \text{ where } w_k \text{ is the weight of } e_k\}$ (i.e., the data transfer time) can be decreased. To the best of our knowledge, it is still open to devise an efficient algorithm to minimize both of the overall redistribution time and communication steps.

Unlike existing algorithms, the main idea behind our work is to partition large data segments into multiple small data segments and properly schedule them in different communication steps without increasing the number of total communication steps. For example, Figure 6 depicts a redistribution graph with the maximum degree Δ=4.

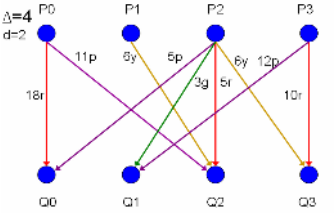


Fig. 6. A redistribution graph with Δ=4

We need four communication steps for this data redistribution since $\chi'(G)=\Delta(G)=4$. In addition, the overall cost of the corresponding scheduling is 38 (See Table 1).

Table 1. The scheduling corresponds to the edge coloring in Figure 6

Step	1(red)	2(yellow)	3(green)	4(purple)	Total
Cost	18	6	3	11	38

Note that the time cost of Step 1 (colored in red) is dominated by the data segment (with 18 data elements) from P_0 to Q_0. Suppose that we evenly partition the segment into two data segments (with 9 and 9 data elements respectively) and transmit them in different steps; then the time required for Step 1 is reduced to 10 (dominated by the data segment from P_3 to Q_3). Note that the data partition adds an edge (P_0, Q_0) in the original redistribution graph. Similarly, we can partition any large data segment into multiple small data segment if the maximum degree of the resulting redistribution graph remains unchanging. After several data partitions, the overall communication cost can be reduced to 29 and the number of required communication step is still minimized (see Figure 7 and Table 2).

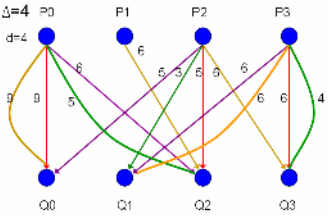


Fig. 7. The resulting redistribution graph after partitioning long data segments

Table 2. The scheduling after partition long data communications

Step	1(red)	2(yellow)	3(green)	4(purple)	Total
Cost	9	9	5	6	29

The idea stated above can be implemented by three major steps: the selection step, the scheduling step, and the partitioning step. The details of the steps will be described in the following subsections.

4.2 Selection Step

In the selection step, we select large data segments for further partition. Each selected data segment introduces one (or more) new dummy edge to the redistribution graph. Suppose there are v_k dummy edges added on an edge e_k, the estimated size of data segments is assume to be $w_k/(1+v_k)$. Note that some large data segments may be divided into more than two segments by adding more than two dummy edges. However, such selections must not increase the number of required communication steps by sustaining the maximum degree Δ of the resulting redistribution graph.

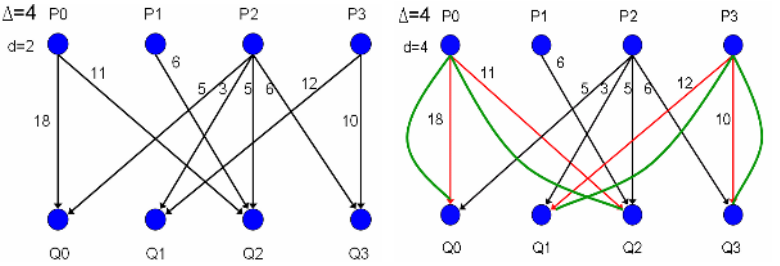


Fig. 8. (a) A redistribution graph (b) The resulting graph after the selection step

The algorithm of the selection step is shown as follows.

Algorithm Selection()
Input: A redistribution graph $G=(S, T, E)$ with maximum degree Δ.
Output: A redistribution graph $G=(S, T, E \cup D)$ with maximum degree Δ, where D represents those dummy edges added in the algorithm.

Step 1. Select the edge $e_k=(s_i, t_j)$ from E such that the value $w_k/(1+v_k)$ is the largest and $d_G(s_i)<\Delta$ and $d_G(t_j)<\Delta$, where v_k denotes the number of added dummy edge with the same end points of e_k. If no such edge exists, terminate this algorithm.
Step 2. Add a dummy edge $e_{k'}=(s_i, t_j)$ to D and set $v_k=v_k+1$.
Step 3. Go to Step 1.

The time complexity of Selection is $O(m \log m)$, where m is the size of edge set of the input redistribution graph.

4.3 Scheduling Step

In the scheduling step, we schedule these data segments (including dummy segments) in the minimum number of communication steps by coloring the edges of its redistribution graph. Since redistribution graphs are bipartite graph, we may apply Cole and Hopcroft's $O(m \log n)$ bipartite edge coloring algorithm [23] (where m, n is the number of edge and vertex, respectively).

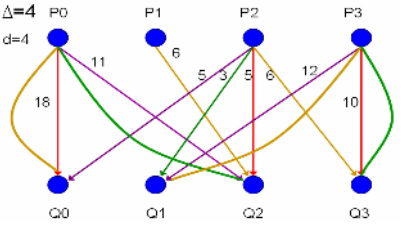


Fig. 9. The bipartite multi-graphs with its colored edges

If the input redistribution graph is restricted to subclasses of bipartite graphs, the scheduling step can be implemented in a more efficient way. Since the redistribution graph $G_{GB(P, Q)}$ of irregular redistribution GEN_BLOCK $GB(P, Q)$ is a biplanar graph,

the resulting graph after the selection step (by adding some dummy edges) is a biplanar multi-graph. As anticipated, the next algorithm will optimally color the edges of G in $O(m=|E|)$ time.

Algorithm Scheduling ()

Input: A biplanar multi-graph $G=(S, T, E)$ with orderings of $S=\{s_0, s_1, \ldots, s_{|s|-1}\}$ and $T=\{t_0, t_1, \ldots, t_{|t|-1}\}$ have its biplanar drawing.

Output: An edge coloring of G with the minimum number of colors Δ.

Step 1. Assign each edge (s_i, t_j) with the integer $(i+j)$.

Step 2. Sort all edges according to the assigned integers in ascending order.

Step 3. Color all edges with Δ colors in turns, according to the order obtained in Step 2.

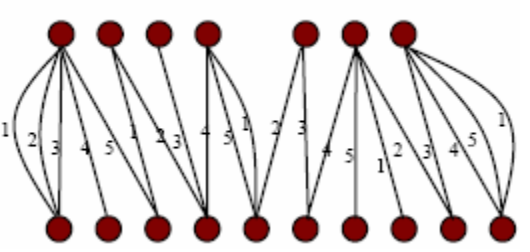


Fig. 10. Coloring the doubly convex-bipartite graphs without crossing edges

Figure 10 presents an example. If G is a doubly convex-bipartite graph, algorithm Scheduling properly color G with its edge chromatic number. Evidently, the time complexity of Scheduling is $O(m)$, where m is the size of edge set of G.

4.4 Partition Step

Let edge colorings $\{E_1, E_2, E_3, \ldots, E_\Delta\}$ of G be the output of Scheduling step. Define C_i to be $\max\{w_k | e_k \in E_i - F$ where w_k is the weight of $e_k\}$; thus the overall redistribution time equals $\sum_{i=1}^{i=\Delta} C_i$ if $F=\varnothing$). Partition step properly partitions and distributes the weights of selected edges (denoted by F) to dummy edges (added in Selection step); thus the final redistribution time is shortened. The algorithm is listed as follows.

Algorithm Partition.

Input: A redistribution graph $G=(S, T, E\cup D)$ with maximum degree Δ, where D represents those dummy edges added in Selection algorithm.

Output: A near optimal total redistribution time.

Step 1. Sort the edges of F increasingly according to their weights.

Step 2. Compute C_i for i=1 to Δ.

Step 3. Select $e_k=(s_i, t_j)$ to be the edge with the lightest weight w_k in F. Let z denote the size of edge set comprising e_k and dummy edges (s_i, t_j) added in Selection step. Suppose that these dummy edges with its associated edge e_k are scheduled in $E_{o(1)}, E_{o(2)}, \ldots, E_{o(z)}$, respectively.

Step 4. If (w_k- $\sum_{K=1}^{K=z} C_{o(K)}$)≤0 then even out weight w_k over those dummy edges by performing {*temp*= w_k; *i*=1;
while *temp*> C_i do {assign the value of C_i to the dummy edge which is scheduled in $E_{o(i)}$; *temp*=*temp*- C_i; *i*=*i*+1;}
If temp>0, assign *temp* to the dummy edges scheduled in $E_{o(i)}$.}
Otherwise, assign the value of C_i+(w_k- $\sum_{K=1}^{K=z} C_{o(K)}$)/*z* to each dummy edge, for *i*=1 to *z*.
Step 5. Delete e_k from *F*;
Step 6. If *F*≠∅ go to Step 2.

For example, the final redistribution graph after Partition step is depicted below.

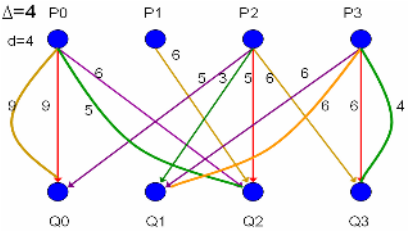


Fig. 11. The resulting redistribution graph after Partition step

The time complexity of Algorithm partition is $O(\Delta m)$ where Δ is the maximum degree of the distribution graph and *m* is the number of its edges. The time complexity of the whole proposed algorithm is $O(m\log m+\Delta m)$.

5 Simulation Results

Our simulations were conducted in C for `GEN_BLOCK` distribution. Given an irregular array redistribution on *A*[1:*N*] over *P* processors, the average size of data blocks is *N*/*P*. Let $T_b(T_a)$ denote the total redistribution cost without (with) applying our algorithm. The *reduction ratio R* equals $(T_b\text{-}T_a)/T_b$. Moreover, let $\{E_1, E_2, E_3, \ldots, E_\Delta\}$ of *G* denote the output of Scheduling step. We also define $C_i = \max\{w_k \mid e_k = (u,v) \in E_i \text{ and either } d(u)=\Delta \text{ or } d(v)=\Delta, \text{where} w_k \text{ is the weight of } e_k\}$. As a result, the overall redistribution time is bounded by $B=\sum_{i=1}^{i=\Delta} C_i$ since the proposed algorithm does not select maximum-degree edges for further partition. Otherwise, the required communication step will be increased.

To thoroughly evaluate how our algorithm affects the data transfer cost, our simulations consider different scenarios. Each data point in the following figures represents an average of at least 10000 runs in each different scenario.

The first scenario assumes that the size of data array is fixed, i.e., *N*=100; the number of processors range from 4, 8, 16, 32, 64, to 128; the size of data blocks is randomly

selected between 1 and 50. In Figure 12, the value of T_b drastically raises as the number of processors increases. However, after applying our algorithm, the overall distribution time T_a smoothly raises as the number of processors increases. Note that the B value drops as the number of processor increase due to the decrease of the average values of data elements in a single communication. In short, when the number of processors increases, the reduction ratio R raises if applying our partition algorithm.

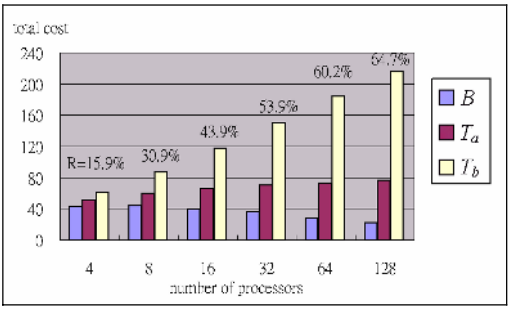


Fig. 12. Simulation results of Scenario I

The second scenario assumes that the number of processors is fixed, i.e., P=32; the size of data array N equals 1600, 3200, 6400, 9600, or 12800; and the size of data blocks is randomly selected between 1 and 2×(N/P). As shown in Figure 13, the values of T_a, T_b, and B raises as the size of data array N increases due to the increase of the average number of data elements in a single communication. However, the reduction ratio stays about 52% by applying our partition algorithm, even with the large size of data array.

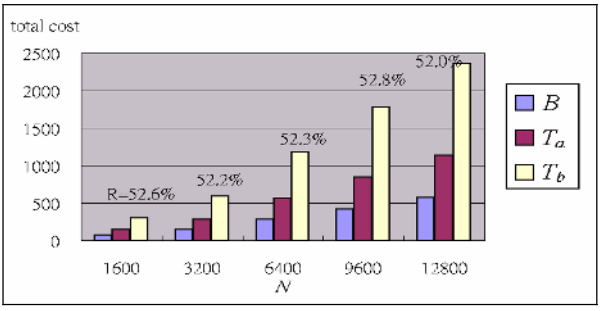


Fig. 13. Simulation results of Scenario II

Scenario III assumes that the size of data array is fixed, i.e., N=3200; the size of data blocks is randomly selected between 1 and 2×N/P; the number of processors varies from 4, 8, 16, 32, 64, to 128. Since the expected size of data blocks is down as the number of processors increases, the resulting total distribution cost is decreased. As a result, the reduction ratio drops as the number of processors increases (Figure 14).

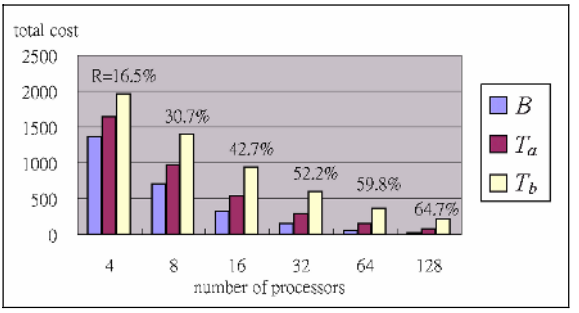


Fig. 14. Simulation results of Scenario III

Scenario IV assumes that both of the size of data array and the number of processors are fixed, i.e., N=3200 and P=32. The size of data blocks is selected randomly between 1 and a given upper bound, which is 150, 200, 400, 800, 1200, or 1600. Since the size of most data blocks is correlated to the given upper bound, the values of T_a, T_b, and B raise as the size of the upper bounds enlarges (See Figure 15). However, when the upper bounds range from 200 to 600, our algorithm owns higher reduction ratio (i.e. R≥50%) (Table 3). As the upper bounds pass and away from 600, the reduction ratio begins to drop.

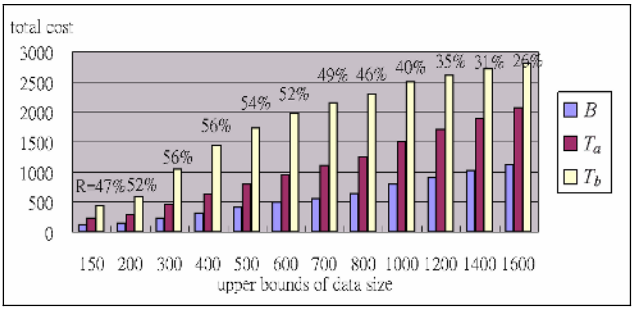


Fig. 15. Simulation results of Scenario IV

Table 3. Reduction ratios with respect to upper bounds

up_bounds	150	200	300	400	500	600	700	800	1000	1200	1400	1600
R	47.01%	52.21%	56.44%	56.10%	54.28%	51.86%	48.88%	45.65%	40.21%	34.70%	30.55%	26.35%

Scenario V assumes that both of the size of data array and the number of processors are fixed, i.e., N=3200 and P=32; the size of data blocks ranges between 1 and 400; the average node degree is 1.95. The maximum degree of the distribution graph ranges from 5 to 15. An edge of end vertex with degree Δ will not be selected to partition data in our algorithm; therefore, the weight of the edge has no possibility to reduce furthermore. On the other hand, an edge of end vertex with degree less than Δ will be

selected for data partition. In Figure 16, the total redistribution cost rises as the maximum degree increases if without applying our algorithm. If our algorithm is applied, however, the total redistribution cost drop slightly.

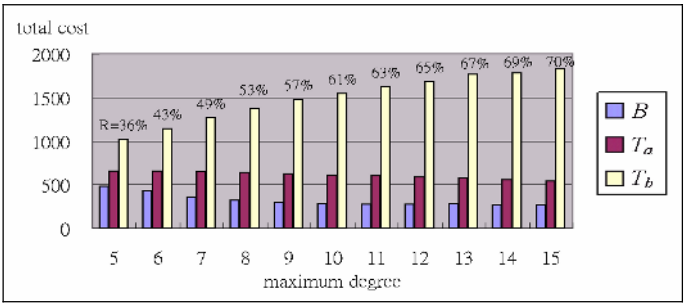


Fig. 16. Simulation results of Scenario V

At last, the following table summaries our simulation results.

Table 4. The main results of our simulations

P	N	N/P	Upper bounds	Δ	R
↑	↑	25	2×(N/P)	random	↑
32	↑	↑	2×(N/P)	random	≥52%
↑	3200	↓	2×(N/P)	random	↑
32	3200	100	2×(N/P) ~ 6×(N/P)	random	≥52%
32	3200	100	4×(N/P)	↑	↑

6 Conclusions and Remarks

We have presented an efficient algorithm to reduce the overall redistribution time by applying data partition. Simulation results indicates that when the number of processors or the maximum degree of the redistribution graph increases or the selected size of data blocks is appropriate, our algorithm effectively reduce the overall redistribution time. In future, we try to estimate the reduction ratio precisely. We also believe that the techniques developed in the study can be applied to resolve other scheduling problems in distribution systems.

References

[1] G. Bandera and E.L. Zapata, "Sparse Matrix Block-Cyclic Redistribution," *Proceeding of IEEE Int'l. Parallel Processing Symposium* (IPPS'99), San Juan, Puerto Rico, April 1999.

[2] Frederic Desprez, Jack Dongarra and Antoine Petitet, "Scheduling Block-Cyclic Data redistribution," *IEEE Trans. on PDS*, vol. 9, no. 2, pp. 192-205, Feb. 1998.

[3] C.-H Hsu, S.-W Bai, Y.-C Chung and C.-S Yang, "A Generalized Basic-Cycle Calculation Method for Efficient Array Redistribution," *IEEE TPDS*, vol. 11, no. 12, pp. 1201-1216, Dec. 2000.

[4] C.-H Hsu, Dong-Lin Yang, Yeh-Ching Chung and Chyi-Ren Dow, "A Generalized Processor Mapping Technique for Array Redistribution," *IEEE Transactions on Parallel and Distributed Systems*, vol. 12, vol. 7, pp. 743-757, July 2001.
[5] Minyi Guo, "Communication Generation for Irregular Codes," *The Journal of Supercomputing*, vol. 25, no. 3, pp. 199-214, 2003.
[6] Minyi Guo and I. Nakata, "A Framework for Efficient Array Redistribution on Distributed Memory Multicomputers," *The Journal of Supercomputing*, vol. 20, no. 3, pp. 243-265, 2001.
[7] Minyi Guo, I. Nakata and Y. Yamashita, "Contention-Free Communication Scheduling for Array Redistribution," *Parallel Computing*, vol. 26, no.8, pp. 1325-1343, 2000.
[8] Minyi Guo, I. Nakata and Y. Yamashita, "An Efficient Data Distribution Technique for Distributed Memory Parallel Computers," *JSPP'97*, pp.189-196, 1997.
[9] Minyi Guo, Yi Pan and Zhen Liu, "Symbolic Communication Set Generation for Irregular Parallel Applications," *The Journal of Supercomputing*, vol. 25, pp. 199-214, 2003.
[10] Edgar T. Kalns, and Lionel M. Ni, "Processor Mapping Technique Toward Efficient Data Redistribution," *IEEE Trans. on PDS*, vol. 6, no. 12, December 1995.
[11] S. D. Kaushik, C. H. Huang, J. Ramanujam and P. Sadayappan, "Multiphase data redistribution: Modeling and evaluation," *Proceeding of IPPS*'95, pp. 441-445, 1995.
[12] S. Lee, H. Yook, M. Koo and M. Park, "Processor reordering algorithms toward efficient GEN_BLOCK redistribution," *Proceedings of the ACM symposium on Applied computing*, 2001.
[13] Y. W. Lim, Prashanth B. Bhat and Viktor and K. Prasanna, "Efficient Algorithms for Block-Cyclic Redistribution of Arrays," *Algorithmica*, vol. 24, no. 3-4, pp. 298-330, 1999.
[14] Neungsoo Park, Viktor K. Prasanna and Cauligi S. Raghavendra, "Efficient Algorithms for Block-Cyclic Data redistribution Between Processor Sets," *IEEE Transactions on Parallel and Distributed Systems*, vol. 10, No. 12, pp.1217-1240, Dec. 1999.
[15] Antoine P. Petitet and Jack J. Dongarra, "Algorithmic Redistribution Methods for Block-Cyclic Decompositions," *IEEE Trans. on PDS*, vol. 10, no. 12, pp. 1201-1216, Dec. 1999.
[16] L. Prylli and B. Tourancheau, "Fast runtime block cyclic data redistribution on multiprocessors," *Journal of Parallel and Distributed Computing,* vol. 45, pp. 63-72, Aug. 1997.
[17] S. Ramaswamy, B. Simons, and P. Banerjee, "Optimization for Efficient Data redistribution on Distributed Memory Multicomputers," *Journal of Parallel and Distributed Computing,* vol. 38, pp. 217-228, 1996.
[18] Akiyoshi Wakatani and Michael Wolfe, "Optimization of Data redistribution for Distributed Memory Multicomputers," short communication, *Parallel Computing*, vol. 21, no. 9, pp. 1485-1490, September 1995.
[19] Hui Wang, Minyi Guo and Daming Wei, "Divide-and-conquer Algorithm for Irregular Redistributions in Parallelizing Compilers", *The Journal of Supercomputing*, vol. 29, no. 2, 2004.
[20] Hui Wang, Minyi Guo and Wenxi Chen, "An Efficient Algorithm for Irregular Redistribution in Parallelizing Compilers," *Proceedings of 2003 International Symposium on Parallel and Distributed Processing with Applications*, LNCS 2745, 2003.
[21] H.-G. Yook and Myung-Soon Park, "Scheduling GEN_BLOCK Array Redistribution," *Proceedings of the IASTED International Conference Parallel and Distributed Computing and Systems*, November, 1999.
[22] J.A. Bondy and U.S.R. Murty, *Graph Theory with Applications*, Macmillan, London, 1976.

[23] R. Cole and J. Hopcroft, "On edge-coloring bipartite graphs," *SIAM J. Comput.* vol. 11, pp. 540-546, 1982.

[24] C. W. Yu and G. H. Chen, "Efficient parallel algorithms for doubly convex-bipartite graphs," *Theoretical Computer Science*, vol. 147, pp. 249-265, 1995.

[25] P. Eades, B. D. McKay, and N. C. Wormald, "On an edge crossing problem," *Proc. 9th Australian Computer Science Conference*, Australian National University, 1986, pp. 327-334.

[26] N. Tomii, Y. Kambayashi, and Y. Shuzo, "On planarization algorithms of 2-level graphs," *Papers of tech. group on electronic computers, IECEJ*, EC77-38, pp. 1-12, 1977.

[27] C. W. Yu, 'On the complexity of the maximum biplanar subgraph problem,' *Information Science*, vol. 129, pp. 239-250, 2000.

Making Power-Efficient Data Value Predictions

Yong Xiao, Xingming Zhou, and Kun Deng

National Laboratory for Parallel & Distributed Processing,
Changsha, P.R. China
yxiao1977@hotmail.com

Abstract. Power dissipation due to value prediction is being more studied recently. In this paper, a new cost effective data value predictor based on a linear function is introduced. Without the complex two-level structure, the new predictor can still make correct predictions on some patterns that can only be done by the context-based data value predictors. Simulation results show that the new predictor works well with most value predictable instructions. Energy and performance impacts of storing partial tag and common sub-data values in the value predictor are studied. The two methods are found to be good ways to build better cost-performance value predictors. With about 5K bytes, the new data value predictor can obtain 16.5% maximal while 4.6% average performance improvements with the SPEC INT2000 benchmarks.

1 Introduction

Data value prediction is a speculative approach that has been widely studied to break true data dependences in programs. By correctly predicting one instruction's resulting value, later dependent instructions can get executed earlier. Thus higher ILP may be obtained. Several value prediction schemes have been proposed [1, 2, 4, 5, 9]. Promising performance benefits have also been reported.

As traditional value predictors mostly maintain one single structure, the high percentage of instructions eligible for value prediction makes the access latency and power consumption a big challenge [11]. Making a power-efficient value predictor is the main focus of many previous studies [6-13, 22, 23].

Stride data value predictors (SVP) and context-based data value predictors (CVP) are two main kinds of value prediction schemes used by the researchers. A stride data value predictor predicts the value by adding the most recent value to a stride, namely the difference of the two most recently produced results. A context-based predictor predicts the value based on the repeated pattern of the last several values observed. It consists of two-level tables: a value history table (VHT) and a pattern history table (PHT). The VHT stores several distinct values that are generated by an instruction and keeps track of the order in which they are generated. Several saturating counters maintained in PHT are used to decide whether a prediction should be made or not. SVP works well with patterns such as (1,2,3,4…) and CVP can make correct predictions with patterns such as (1,6,2,5,1,6,2,5…). We can see that on one side CVP can reap

T. Srikanthan et al. (Eds.): ACSAC 2005, LNCS 3740, pp. 310 – 322, 2005.

additional performance benefits by predicting data sequences with complex patterns. On the other side, CVP is more complex than SVP. Better tradeoffs should be made for better design decisions.

In this paper, a new kind of value predictor based on a linear function is proposed. With some modifications on SVP, the new predictor can make correct predictions on some patterns (e.g. 1,-5,1,-5…) that can only be done by CVP. Simulation results show that the predictor works well with most value predictable instructions. Energy and performance impacts of storing partial tag and common sub-data values in the value predictor tables are studied. The two methods are found to be good ways to build better cost-performance value predictors. With about 5K bytes, the new data value predictor can obtain 16.5% maximal while 4.6% average performance improvements with the SPEC INT2000 benchmarks.

The remainder of the paper is organized as follows: Section 2 discusses recent relevant work in data value prediction. Section 3 introduces the new value prediction scheme. Section 4 summarizes our experimental methodology. Section 5 presents performance and cost comparisons with other value predictors. Section 6 analyzes the energy and performance impacts of storing partial tag and common sub-data values in the value predictor tables. The last section presents a summary of this work.

2 Related Work

According to the study of data value locality, several value prediction schemes have been proposed [1-4]. These schemes can be broadly classified into three categories: (1) computation-based predictors, including last value predictors (LVP) [1, 2] and stride data value predictors (SVP) [4]. The predictor that exploits global stride value locality [15] also belongs to this type; (2) context-based predictors [3, 4]; (3) hybrid predictors, such as the stride+2level hybrid predictor (S2P) [4, 9]. The stride+2level hybrid value predictor is a combination of SVP and the context-based predictor (CVP). CVP is always accessed first, but its prediction is used only if the maximum counter value is greater than or equal to the threshold value. Otherwise, SVP is used. With more complexity, hybrid predictors achieve higher prediction capability. In [17], the performance impacts of these predictors are discussed. SVP and S2P are found to have better cost performance.

The power aspect of value prediction is gaining more attention recently. For a power efficient value prediction implementation, designers can either limit the predictor's complexity [8, 9, 13, 22, 23] or limit the number of access ports [6, 7, 10-12]. Moreno et al [8,9] explore the main sources of power dissipation to be considered when value speculation is used, and propose solutions to reduce this dissipation – reducing the size of the prediction tables, decreasing the amount of extra works due to speculative execution and reducing the complexity of the out-of-order issue logic. In [13], the value prediction table is partitioned into several smaller structures using data-widths. T. Sato et al. [22, 23] propose using partial resolution to make value prediction power-efficient. Selecting value prediction candidates, such as choosing instructions on the critical path [6, 7] for value prediction reduces the amount of accesses to the value predictor and causes smaller power consumptions.

Y. Sazeides proposes a methodical model for dynamically scheduled microarchitectures with value speculation [16]. In the paper, the model adopted by us [17] is used for the experiments.

3 Scheme for Linear Function Based Predictor

Theoretically, LVP and SVP can be thought of using equations EqLt: ($V_{pred} = V_{last}$) and EqSt: ($V_{pred} = V_{last} + V_{stride}$) respectively to make their predictions. Both equations are instances of the linear function EqLi: ($V_{pred} = Cm * V_{last} + Ca$), where Cm and Ca are calculated coefficients. Obviously schemes that use EqLi to make predictions have some advantages over last value and stride value predictions: (1) those values predictable by LVP or SVP can also be predicted by EqLi, (2) some patterns predictable by CVP can also be predicted by EqLi, such as sequences of the form (3,-5,3,-5...) where Cm = -1 and Ca = -2. For SVP such sequences can not be predicted.

Direct implementation of EqLi is inefficient due to the complex calculations of both the predicted results and the coefficients. Restrictions should be put on the two coefficients. Simulation analysis shows that on average more than 96%[1] of the correct predictions made by EqLi fall into the four types: (1) Cm=0, (2) Cm=1 and Ca=0[2], (3) Cm=1 and Ca≠0, (4) Cm=-1.So the linear function based value prediction scheme (LFP) can be much simplified.

3.1 Block Diagram of LFP

In our implementation, one load instruction is divided into two operations: load address calculation and memory access. Thus LFP has two parts: the load address predictor and the load/ALU predictor. Figure 1 shows LFP's block diagram.

In LFP, field Tag holds the whole or partial address of one instruction. Field S is used for type (3) prediction mentioned above. When S is set to 1, field V0 will be used to hold the stride, and only field V1 will be updated with the latest produced data by one instruction. Otherwise, field V0 and V1 are updated alternatively to hold the two most recently produced result. Field W indicates which the latest one is. Three saturating counters, namely CL, CS and CR are used in each value history table (VHT) entry. CL, CS and CR are used for prediction type (1) or (2) (e.g. 4,4,4,4…), type (3) (e.g. 1,2,3,4…) and type (4) (e.g. 1,-5,1,-5…) respectively. The three counters can be set with different priorities, which will result in different performance gains. As type (1) or (2) needs the shortest learning time before making correct predictions, while type (4) needs the longest, in our implementation, CL is set with the highest priority while CR the lowest. The Threshold is used to control the misprediction rate under an acceptable level.

---

[1] Due to space limits, we do not present the experiment results here.

[2] Type (1) and (2) are in fact identical, but for our statistic process, sequences of the form (1,4, 4,4 …) will be classified as type (1), while sequences of the form (4,4,4 …) will be classified as type (2).

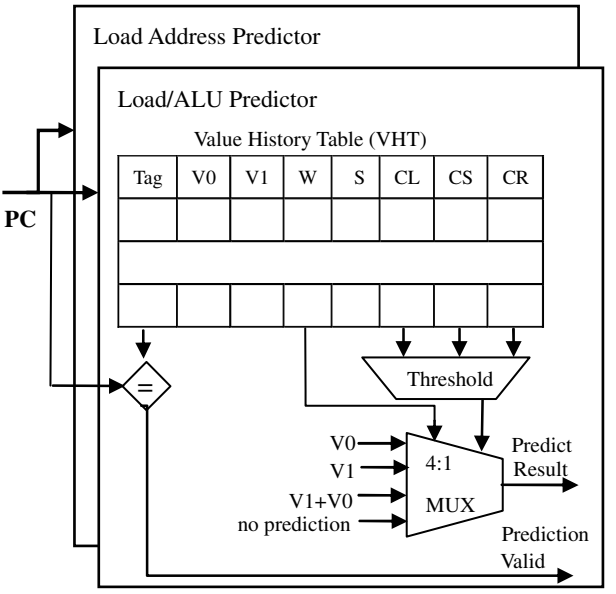


Fig. 1. Linear Function based Data Value Predictor

3.2 LFP's Working Process

The LFP's lookup process works as follows:

step 1. At dispatch stage, the load/ALU predictor table is accessed with the instruction's address (PC). If the instruction is a load instruction, the PC is also used to access the load address predictor table

step 2. The Tag is compared with the PC

step 3. If there is a match, go to step 4; otherwise, no prediction is made

step 4. Three counters are compared with the specific threshold value.

step 5. In the order of CL, CS and CR, the first counter that is bigger or equal to the threshold value is used for value prediction

step 6. If one counter's value is bigger or equal to the threshold value, then its corresponding value is returned; else, no prediction is made

For example, assume that field W=1, LFP will return V1, V1+V0 or V0 respectively in case that the selected counter is CL, CS or CR. If all count values are less than the threshold, then no prediction is made. Once a prediction is made, later dependent instructions will be able to get issued with the predicted value. In the paper, LFP is immediately updated with the result after the lookup process.

LFP is updated as bellow. The count value corresponding to the correct outcome is incremented by *I*, and all other count values are decremented by *D*. If tag mismatch happens, one new entry will be allocated for the new instruction. Sometimes this will cause one old entry to be dropped if the new entry allocated is not empty.

The confidence mechanism which can be represented by a tuple <*I, D, T, S*> has important impact on the performance. It includes the design of *I*, *D*, the threshold (*T*) and the counter saturation value (*S*). In the paper, we will use the mechanism of <2, 1, 2, 3> and <1, 1, 2, 3> for the load/ALU predictor and the load address predictor respectively. It is found to have the best performance.

In our implementation, value prediction is only performed for load and ALU instructions. Branch instructions, store instructions etc are not considered. Except store operations, other instructions can all get executed speculatively.

4 Experiment Environment

4.1 Benchmarks

The SPEC CINT2000 [21] benchmark suite is used[3]. All benchmarks are compiled with the gcc-2.6.3 compiler in SimpleScalar [18] (PISA version) with optimization flags "–O2 -funroll-loops". The reference input sets are used. Simulation points are chosen according to [24-26] and the parameter "early" is used. Table 1 shows the benchmarks, the specific input sets and the simulation points (SM) used.

Table 1. Benchmarks, Input Sets and Simulation Points

SPEC INT2000	Input Set	SM (100M)
164.gzip	input.source 60	156
175.vpr	net.in archi.in place.out dum.out $FLAGS	543
176.gcc	scilab.i –o scilab.s	10
181.mcf	inp.in	25
197.parser	2.1.dict –batch < ref.in	19
255.vortex	lendian1.raw	123
256.bzip2	input.graphic 58	13
300.twolf	ref	11[4]

4.2 Architecture Parameters

SimpleScalar's default latency for each functional unit is used. Other simulation configurations are summarized in Table 2.

[3] Only eight benchmarks are used. Other benchmarks are not available for PISA SimpleScalar.

[4] *Twolf* runs into an unimplemented syscall call with SimpleScalar at about 1500M instructions. Thus we do not choose its simulation point using the SimPoint. Instead, we skip the first billion instructions and simulate the next 100M instructions.

Table 2. Architecture Parameters

L1 instruction cache	32KB, directly-mapped, block size 32B
L1 data cache	256KB, 4-way, block size 32B, 3 cycles hit latency
L2 instruction/data cache	1MB, 8-way, block size 64B, 12 cycles hit latency
Memory	250 cycles for the first 8B, 2 cycles for each next 8B
TLB	16 entries instruction TLB, 32 entries data TLB, both TLB are 4-way associative, 30 cycles miss penalty
BTB	512 entries, 4-way
RAS (return address stack)	16 entries
Instruction window	96
Load/store queue	48
Fetch/decode/issue/commit width	4/4/6/6 instructions/cycle
Functional unit numbers	int alu:4 int mult/div:1 fp adder:2 fp mult/div:1 memory ports:2
Branch predictor	bimodal, 8K entries

5 Experimental Results

In this subsection, we first show the performance impact of different LFP table sizes. Then performance comparison as well as hardware cost and power consumption aspects of different predictors are presented. The instruction per cycle (IPC) of the base microarchitecture without value prediction implementation is shown in Table 3.

Table 3. Base IPC for Each Benchmark

	bzip2	gcc	mcf	parser	twolf	vortex	vpr	gzip
IPC	0.7495	0.9916	0.1433	1.1361	0.7160	1.3157	1.0894	1.6315

5.1 Performance Impact of Predictor Table Size

Performance impact of different predictor table sizes is discussed in this subsection. As load instructions occupy about one half of all instructions that are eligible for data value prediction, the size ratio of the load/ALU predictor to the load address predictor is then set to 2 in later experiments. For example, when the load/ALU predictor is set to 2K entries, the load address predictor is set to 1K entries.

Figure 2 presents the results. The size labeled in the figure stands for the load/ALU predictor's size in K-entry. As instructions will occupy different entries, generally saying, the performance will be improved when bigger size predictors are used. For *vpr*, when the predictor size is increased from 256-entry to 64K-entry, the speedup improves from 2.6% to 4.7%. From Figure 2, only diminishing improvements can be obtained for most benchmarks as predictor table size increases. Thus smaller predictors are good choices for power-efficient value predictor design.

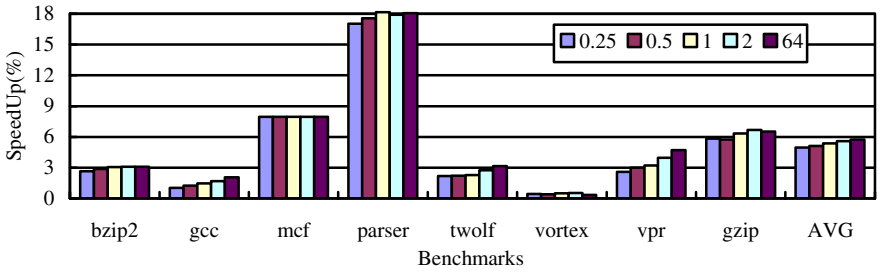


Fig. 2. Performance Impact of Different Table Size (K entries)

5.2 Performance and Cost Comparisons of the Data Value Predictors

Performance and hardware costs of the predictors, namely stride value predictor (SVP), stride+2level hybrid predictor (S2P) and LFP are studied in the subsection. SVP and S2P realized by Sang [20] are referenced for comparison.

5.2.1 Performance Comparison of the Predictors

The block size for LFP is calculated as below: each entry will need 4 bytes (Tag) + 4 bytes * 2 (V0 and V1) + 2 bits (W and S) + 2bits * 3 (CL, CS and CR), namely 13 bytes. Similarly, SVP and S2P's block sizes are 13 bytes and 26 bytes respectively.

Instructions eligible for value predictions are more than 60% in one program [17], thus in one cycle, many instructions need to access the predictor. To keep the CPU frequency high and to control the power consumption and the design complexity to an acceptable level, the predictor cannot be too large. In the simulation, the predictor size is set as below: (1) LFP's load/ALU predictor and load address predictor are 2K entries and 1K entries respectively, (2) SVP's are 2K entries and 1K entries respectively, (3) S2P's are 1K entries and 512 entries respectively, and their pattern history tables are both 2K entries. So each kind of predictor needs about 39K bytes.

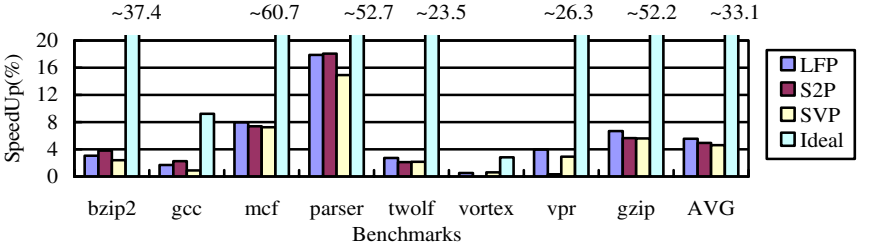


Fig. 3. Performance Comparison of Different Data Value Predictors

Figure 3 presents the performance comparisons. We also present ideal value prediction's performance improvements. Ideal value prediction means that the predictions are always correct and data dependent instructions can always get issued with correct inputs if no other hazards happen. With ideal prediction, we can know how much benefits at most can be obtained by data value prediction. We can see that for most benchmarks, except *gcc* and *vortex*, ideal data value prediction can bring great performance gains. The average improvement for ideal data value prediction is about 33.1%. The low benefits with *gcc* and *vortex* indicate that the performances of these two programs' simulated segments are not limited by data hazards.

For *gcc* and *bzip2*, S2P performs the best among all predictors. The negligible gain obtained with *vpr* is due to S2P's high reissue rate. The average speedup for S2P is 4.96%. For other programs, LFP performs better than both S2P and SVP. For *parser*, LFP obtains 17.9% speedup, while SVP obtains 14.9%. The average speedup for LFP is 5.57%. Compared with ideal value prediction's performance, the gains obtained by the three predictors are moderate. How to better exploit the potential of data value prediction is a hot topic in current researches.

5.2.2 Cost and Energy Comparisons

This subsection compares the cost and energy aspect of different value predictors. For comparison, CACTI 3.2 [19] is used. CACTI is an integrated cache access time, cycle time, area, aspect ratio, and power model.

In the simulation, the dispatch width is 4. Thus at maximum 4 value predictions need to be made per cycle. In CACTI, 2 read/write ports can be simulated. So in our experiment, 2 read/write ports, 2 read ports and 2 write ports are used. For S2P, only the data of the value history table (VHT) is calculated. Table 4 lists the energy, access time and area aspects of the predictors. All predictors are direct-mapped and no bank mechanism is used. 90nm technology is simulated. It can be seen that LFP and SVP need less hardware costs than S2P. Considering the performances shown in Figure 3, LFP is a good candidate for application.

Table 4. Value Predictor Cost

Predictor	Energy(nJ)	Access Time(ns)	Area(mm^2)
SVP	2.48	1.09	10.98
LFP	2.48	1.09	10.98
S2P	3.41	1.12	11.39

6 Making LFP More Power-Efficient

As studied in [13, 22, 23], with little performance benefits loss, storing partial tag and/or data in the value predictor tables can reduce power consumption. In this section, the effects of storing partial tag and storing common sub-data values in LFP are studied. LFP's size is the same as that in subsection 5.2.1.

6.1 Storing Partial Tag

Performance impact of storing different length partial tags in LFP is shown in Figure 4. Tag lengths vary from 32bits to 0bit[5]. The labels in the figure are in the form of 'A_B', where 'A' stands for the load/ALU predictor's tag length and 'B' stands for the load address predictor's. It can be seen that the length of tags stored in LFP will hardly impact the performance. When the 2K-entry LFP is used, the miss rates of accessing the predictor tables are mostly below 5%. This indicates that instructions will seldom occupy the same entry. So the Tag field in LFP is not so important.

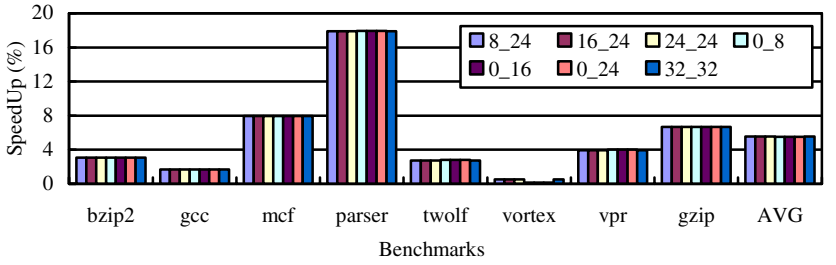


Fig. 4. Performance Impact of Storing Partial Tag

6.2 Storing Common Sub-data Values

Results produced by one instruction often have some parts that contain the same value. We call such value common sub-data value. The bit width of the value is defined as the common sub-data value's length. For example, considering the sequence of (0x10615088, 0x10615108, 0x10615188, 0x10615208…), where the higher 20bits are the same. The common sub-data value is '0x10615', and its length is 20. For such sequences, the method of storing common sub-data values in the tables can avoid the redundant storing and thus save hardware costs. For the above sequence, in each entry of LFP, instead of using 64 bits to store the two 32bits values, only 44 bits are needed, i.e. 20bits for '0x10615' and two 12bits fields for (0x'088', 0x'108', 0x'188', 0x'208'…).

To implement the Storing Common Sub-data values (SCS) method in LFP, one more field, namely SC (Store Common) is added in each entry to store the common sub-data value. Performance impacts of storing different length common sub-data values in LFP are illustrated in Figure 5. Labels in Figure 5 indicate the lengths of the data values (V0/V1) stored in the two LFP component predictors. For example, '8_16' means that in the load/ALU predictor field V0 (V1) occupies 8 bits, while in the load address predictor field V0 (V1) occupies 16 bits. Accordingly the lengths of common sub-data values for the two component predictors are 24 (namely 32-8) and 16 (namely 32-16) respectively.

[5] As the lower bits are used for the index, thus for 1K sets table, storing 19bits will be enough.

It can be seen that when the SCS method is used, most programs' performances are rarely affected by field V0's or V1's length. This indicates that the results produced by most instructions do not change much. So the SCS method is well suited for power efficient value predictor design. One exception is for *mcf* when '32_8' is used. Under such a configuration, *mcf*'s speedup degrades greatly to just about 3%. For *mcf*, the memory accessing addresses of one load instruction vary much, thus the 8bits V0/V1 and the 24bits common sub-data value are inefficient to represent the load addresses' behavior. As a result, a lot of predictable load addresses are not correctly predicted. For power efficient design, in later simulations the '8_16' mechanism will be used, because it works fairly well among all the programs. Its average speedup is 5.16%, which is just a little lower than 5.47% when full data are stored.

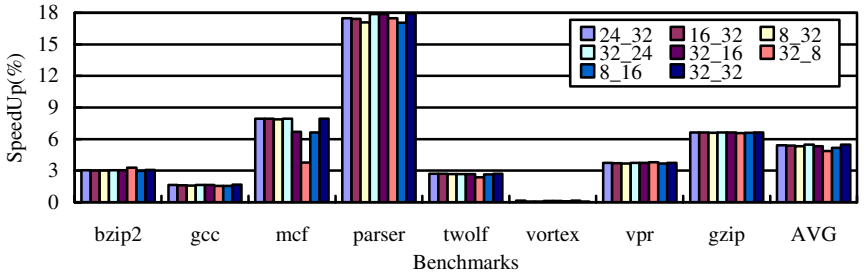


Fig. 5. Performance Impact of Storing Common Sub-Data Values

6.3 Making Power-Efficient Value Predictions

As a conclusion, Figure 6 and Table 5 present the performances and costs of different LFPs. The methods mentioned in the above sections can also be easily implemented within SVP and S2P. The optimal implementation of such methods within SVP and S2P is beyond the scope of the paper.

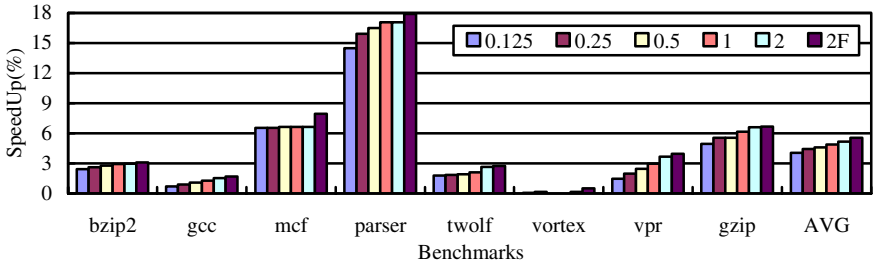


Fig. 6. Performance of Different LFPs

The numbers in the 'Predictor' column of Table 5 stand for the load/ALU value predictors' sizes (in K-entry). 'F' stands for full length data and tag are stored. Except for '2F', all other LFPs adopt the methods of storing partial tag and storing common sub-data values as mentioned in subsections 6.1 and 6.2. It can be seen that as the predictor size increases, the performances of most programs also get improved. Yet beyond the LFP with 512-entry load/ALU predictor and 256-entry load address predictor, only diminishing benefits will be obtained. The LFP that comprises 512-entry load/ALU predictor and 256-entry load address predictor occupies about 4.75K bytes. For such LFP, the biggest speedup obtained is 16.5% (for *parser*) and the average speedup is 4.6%.

Table 5. Different LFPs' Costs

Predictor	Size(KB)	Energy(nJ)	Access Time(ns)	Area(mm^2)
2F	39	2.48	1.09	10.98
2	19	1.19	0.90	5.78
1	9.5	0.89	0.74	3.30
0.5	4.75	0.70	0.58	2.04
0.25	2.375	0.62	0.52	1.29
0.125	1.1875	0.57	0.46	0.78

To make comparisons easier, we define two metrics for illustration: SpkB and SpnJ. SpkB is defined as the speedup obtained per kilobytes. And SpnJ is defined as the speedup obtained per nano-joule. These two metrics explicitly describe how much performance improvement is obtained per unit hardware cost.

The average speedups obtained in Figure 6 combined with the data in Table 5 are used to calculate the SpkB and SpnJ values. Figure 7 shows the SpkB and SpnJ diagrams for all predictors. It can be seen that as LFP size increases, both SpnJ and SpkB become worse, which means that smaller size LFPs have better cost performance. Moreover, the LFP with full tag and data stored has the worst cost performance.

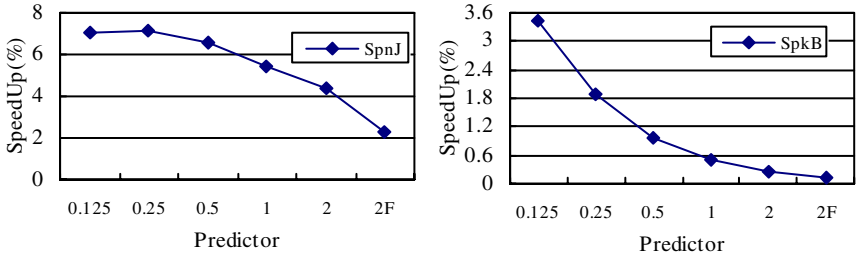


Fig. 7. Resource and Energy Aspect of Different Data Value Predictors

7 Conclusions

In the paper, we have proposed a new cost effective value predictor based on a linear function. The scheme for LFP is introduced. Performance, cost and energy comparisons of LFP, SVP and S2P are made. Results show that LFP is more cost-effective. Moreover, power and performance impacts of storing partial tag as well as storing common sub-data values in the value predictor tables are studied. The methods are found to be good ways to build better cost-performance value predictors. With about 5K bytes, the new data value predictor can obtain 16.5% maximal while 4.6% average performance improvements with the SPEC INT2000 benchmarks.

In future works, the hardware cost and power consumption of the whole system caused by data value prediction implementation are worth studying.

References

1. M.H. Lipasti, J.P. Shen. Exceeding the Dataflow Limit via Value Prediction. Proceedings of 29th International Symposium on Microarchitecture, pp. 226-237, 1996.
2. J.P. Shen, M.H. Lipasti. Exploiting Value Locality to Exceed the Dataflow Limit. International Journal of Parallel Programming 26(4), pp. 505-538, 1998.
3. Y. Sazeides, J. E. Smith. The Predictability of Data Values. Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture, pp.248-258, 1997.
4. K. Wang, M. Franklin. Highly Accurate Data Value Prediction using Hybrid Predictors. Proceedings of the 30th Annual International Symposium on Microarchitecture, pp.281-290, December, 1997.
5. S. J. Lee, Y. Wang, P.C. Yew. Decoupled Value Prediction on Trace Processors. The 6th International Symposium on High Performance Computer Architecture, pp. 231-240, January, 2000.
6. E. Tune, D.N. Liang, D.M. Tullsen, B. Calder. Dynamic Prediction of Critical Path Instructions. The 7th International Symposium on High Performance Computer Architecture, January, 2001.
7. B. Calder, G. Reinman, D.M. Tullsen. Selective Value Prediction. Proceedings of the 26th Annual International Symposium on Computer Architecture, June, 1999.
8. R. Moreno, L. Pinuel, S.D. Pino, F. Tirado. Power-Efficient Value Speculation for High-Performance Microprocessors. Proceedings of the 26th EUROMICRO Conference, September, 2000.
9. R. Moreno, L. Pinuel, S.D. Pino, F. Tirado. A Power Perspective of Value Speculation for Superscalar Microprocessors. Proceedings of the 2000 IEEE International Conference on Computer Design: VLSI in Computers & Processors, 2000.
10. R. Bhargava, L.K. John. Value Prediction Design for High-Frequency Microprocessors. Technical Report TR-020508-01, Laboratory for Computer Architecture, the University of Texas at Austin, May, 2002.
11. R. Bhargava, L.K. John. Latency and Energy Aware Value Prediction for High-Frequency Processors. The 16th International Conference on Supercomputing, pp. 45-56, June 2002.
12. R. Bhargava, L.K. John. Performance and Energy Impact of Instruction-Level Value Predictor Filtering. First Value-Prediction Workshop (VPW1) [held with ISCA'03], pp.71-78, June, 2003.

13. G.H. Loh. Width Prediction for Reducing Value Predictor Size and Power. The First Value-Prediction Workshop (VPW1, Held in conjunction with ISCA-30), June, 2003.
14. F. Gabbay, A. Mendelson. The Effect of Instruction Fetch Bandwidth on Value Prediction. The 25th International Symposium on Computer Architecture, pp. 272-281, 1998.
15. H.Y. Zhou, J. Flanagan, T.M. Conte. Detecting Global Stride Locality in Value Streams. The 30th ACM/IEEE International Symposium on Computer Architecture, June, 2003.
16. Y. Sazeides. Modeling Value Speculation. The 8th International Symposium on High Performance Computer Architecture (HPCA-8), 2002.
17. Y. Xiao, K. Deng, X.M. Zhou. Performance Impact of Different Data Value Predictors. The Ninth Asia-Pacific Computer Systems Architecture Conference, September, 2004.
18. D.C. Burger, T.M. Austin. The SimpleScalar Tool Set, Version 2.0. Technical Report CSTR-97-1342, University of Wisconsin, Madison, June 1997.
19. P. Shivakumar, N.P. Jouppi. CACTI 3.0: An Integrated Cache Timing, Power, and Area Model. WRL Research Report 2001/2, August, 2001.
20. S. J. Lee. Data Value Predictors. http://www.simplescalar.com/.
21. SPEC CPU2000 Benchmarks. http://www.spec.org/osg/cpu2000/.
22. T. Sato, I. Arita. Partial Resolution in Data Value Predictors. Proc. of ICPP, 2000.
23. T. Sato, I. Arita. Table Size Reduction for Data Value Predictors by exploiting Narrow Width Values. Proc. of ICS, 2000.
24. J.J. Yi, S.V. Kodakara, R. Sendag, D.J. Lilja, D.M. Hawkins. Characterizing and Comparing Prevailing Simulation Techniques. 11th International Symposium on High-Performance Computer Architecture (HPCA'05), 2005.
25. T. Sherwood, E. Perelman, G. Hamerly, B. Calder. Automatically Characterizing Large Scale Program Behavior. Tenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2002), October 2002.
26. E. Perelman, G. Hamerly, B. Calder. Picking Statistically Valid and Early Simulation Points. International Conference on Parallel Architectures and Compilation Techniques (PACT 2003), September 2003.

Speculative Issue Logic

You-Jan Tsai and Jong-Jiann Shieh

Department of Computer Science and Engineering,
Tatung University, Taipei, Taiwan
G9206006@ms2.ttu.edu.tw, shieh@ttu.edu.tw

Abstract. In order to enhance the performance of a computer, most modern processors use superscalar architecture and raise the clock frequency. Superscalar architecture can execute more than one instruction per each cycle. The amount of instruction level parallelism will become more and more important when superscalar issue width is increased. With hardware support, instructions can be speculatively waked up. The more instructions are waked up, the more ILP is exploited, hence IPC is increased. Through speculative aspect can be adopted to wakeup more instructions. But the ILP is still limited by the true data dependency. In this paper we proposed the speculative wakeup logic with value prediction mechanism to overcome the data dependency that will exploit instruction level parallelism. And in order to reduce the recovery frequency, we also propose priority-based select logic with two bit counter. In our experiment, our model will enhance performance by 18.02%.

1 Introduction

Dependencies between instructions restrict the instruction-level parallelism (ILP) and make it difficult for the processor to utilize the available parallel hardware. There are two kinds of data dependencies: true data dependencies and false data dependencies (or name dependencies). False data dependencies can be eliminated using hardware and software techniques such as register renaming. But true data dependencies can greatly impede ILP. If an instruction is data dependent on a preceding instruction, then it can be executed only after the preceding instruction's result becomes available. In order to achieve a higher processor clock rate, in the past twenty or more years, pipeline depths have grown from 1 (Intel 286), now up to over 20 (Intel Pentium 4) [8]. There are pieces of logic that must evaluate in a single cycle to meet IPC (Instructions Per Cycle) performance goals.

A high IPC rate implies hardware has to fetch and issue multiple instructions in parallel. The conventional RUU (Register Update Unit) architecture [4] was set up to solve the problem of data and control dependence, and enhances the effectiveness of issue logic.

The issue logic determines when instructions are ready to execute. The issue logic consists of two parts: wake up logic and select logic. Instructions cannot be waked up until all instructions they are dependent on have been executed. Both wakeup and select logic form a critical loop. If this loop is stretched over more than one cycle, dependent instructions cannot execute in consecutive cycles.

T. Srikanthan et al. (Eds.): ACSAC 2005, LNCS 3740, pp. 323 – 335, 2005.

Value prediction can break the data dependence among instructions. Therefore, the performance will be increased because the instruction level parallelism (ILP) was exploited. Instructions could be speculatively executed with the aid of value prediction mechanism while the operands of the instructions are not ready. So, processor can adopt the speculative aspect to wake up instructions to be executed even if the operands of the instructions are still not ready.

In this paper we will introduce a new issue logic to select instructions for execution. It adopts the speculative aspect to used value prediction mechanism wake up more instructions. The select logic using the priority value to select instructions for execution. This way, the effectiveness of issue logic will be improved

The rest of this paper is organized as follows: Section 2 introduces the related works. Section 3 describes the design of the proposed issue logic. Section 4 presents the evaluation methodology, and section 5 analysis the performance. Finally, we summarized this study in section 6.

2 Related Works

2.1 Value Prediction

In the case of value prediction, several schemes have been proposed. They are last value prediction [3], stride value prediction [9], context-based prediction [10,11], and hybrid value prediction [10,11,12]. Last value prediction predicts the result value of an instruction based on its most recently generated value. A stride predictor predicts the value by adding the most recent value to the difference of the two most recently produced values. This difference is called the stride. Context-based predictors predict the value based on the repeated pattern of the last several values observed. FCM [10] and two-level [11] predictors belong to this category. Each of the predictors mentioned above shows good performance for certain data value sequences, but bad for others. Therefore, some hybrid predictors are proposed by combining several predictors. Wang et al. proposed a hybrid predictor which combines a stride predictor and a two-level value predictor [11]. Rychlik et al. [12,13] combine a last value predictor, a stride predictor and a FCM predictor. The choice of a predictor for each instruction is guided by a dynamic classification mechanism.

Liao and Shieh [5] presented the new concept of combining the value prediction and data reuse that may lead to new directions in designing future computer architectures. With this mechanism, they attempted to collapse true-data dependences by performing speculative execution based on both Value Prediction and Value Reuse.

Lee and Yew [2] proposed to augment the trace cache with a copy of the predicted values for data-dependent instructions. It allows predicted values to be accessed easily and avoids the problem of having to access a centralized value prediction table. Actual accesses to the value prediction tables are moved from the instruction fetch stage to a later stage, such as the write-back stage. They use a hybrid value predictor with a dynamic classification scheme which can distribute predictor updates to several behavior-specific tables. It allows further reduction in bandwidth requirement for each particular value prediction table.

2.2 Issue Logic

Stark et al. [14] demonstrates that the dynamic instruction scheduling logic can be pipelined without sacrificing the ability to execute dependent instructions in consecutive cycles. It adopts the speculative aspect to wake up more instructions.

In addition to the wakeup logic, they add the speculative wake up operation as shown in figure 1. After the two sources of an instruction are ready, the Grant signal will drive the Dest Tag in wakeup circuit. There are more than three kinds of situations in Speculative wakeup logic: one, the left source is ready, the parents of right source are ready, but right source is not ready; another, the right source is ready, the parents of left source are ready, but left source is not ready; the other, the right source and left source are not ready, but the parents of left source and the parents of right source are ready. There are more and more instructions waiting for selection by speculative wakeup logic.

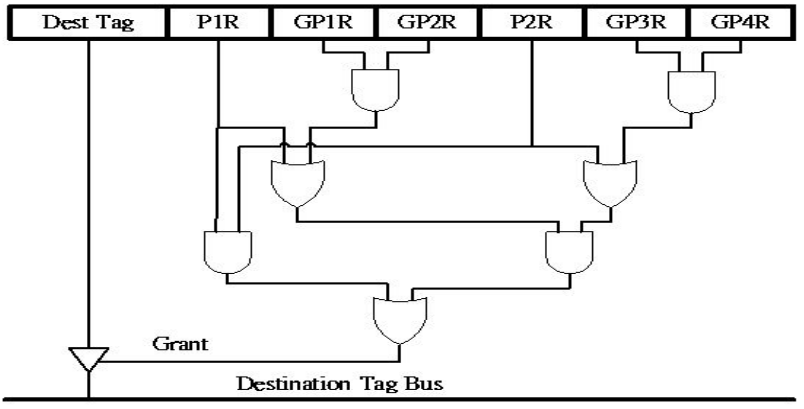


Fig. 1. Speculative Wakeup Logic

Select-Free scheduling logic [7] describes a technique that break the scheduling (wakeup and select) logic into two smaller loop: a critical loop for wakeup and a non-critical for select. With select-free scheduling logic, this will solve the collisions (where more instructions wake up than can be select, resulting in a mis-speculation) and pileups (dependents of the collision victims may wake up before they are really ready to be scheduled, that is entering the scheduling pipeline too early). This paper introduces the select-N schedulers and predicts another wake up (PAW) to avoid the collision.

Issue logic with issue table was proposed by Shiao and Sheih [6]. It uses the wakeup and speculative wakeup logic to enhance the instructions parallelism in data dependency. They propose the issue table to help the select logic selects the suitable instructions to issue. The issue table is divided into 4 levels. When the instruction is wakeuped, the instruction's information is allocated into issue table by means of the level of RUU. The select logic picks instructions with the highest level (issue level 4 table) for execution. If there is no instruction in level 4 table, the select logic goes to the level 3 table and so on.

Ernst et al. present the Cyclone scheduler in [15], a novel design that captures the benefits of both compile-time and run-time scheduling. Once scheduled, instructions are injected into a timed queue that orchestrates their entry into execution. To accommodate branch and load/store dependence speculation, the Cyclone scheduler supports a simple selective replay mechanism.

Ernst et al. [16], it has introduced more efficient reduced-tag scheduler designs that improve both scheduler speed and power requirements. By employing more specialized window structures and last-tag speculation, a large percentage of tag comparisons were removed from the scheduler critical path. These optimizations reduced the load capacitance seen during tag broadcast while maintaining instruction throughputs that are close to those of inefficient monolithic scheduler designs. The optimized designs allow for more aggressive clocking and significantly reduce power consumption.

3 The Design of Issue Logic with Value Speculation

3.1 Instruction's Ready Latency

Recent studies [17, 18] show that predicting the results of instructions which are on the critical-path will improve performance more than on non-critical path instructions. However, deciding the instruction on the critical-path or not is very difficult. It probably costs much hardware and will be very complicated. In this section, we will show the different viewpoint.

To quantify the degree of operand ready latency by reservation stations, a typical 4-wide superscalar processor was simulated using the SimpleScalar toolset [1]. (More details on our experimental framework and baseline microarchitecture model can be found in Section 4.1.)

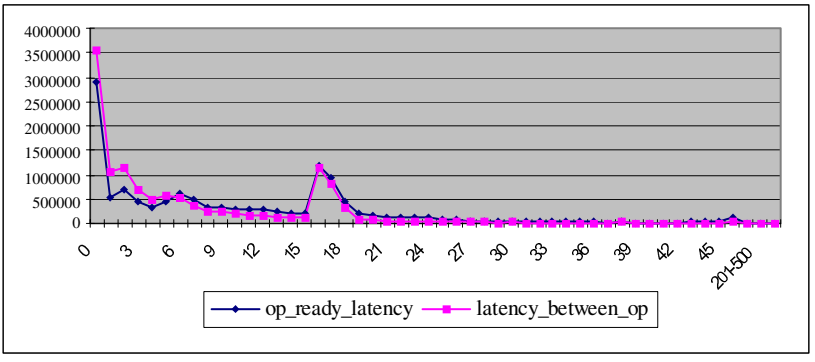


Fig. 2. Instruction Operand Latency

As illustrated in figure 2, that most instructions are waiting for the last operand ready. There are many instructions operand latency between 16 to 20 cycles. That could be depending on the cache miss instructions. If we could predict the instruc-

tion's result that was other instructions' last operand would have much benefit. It didn't need to decide the instruction was on the critical path or not. Therefore it will simplify the implementation.

3.2 Data Value Prediction

Several architectures have been proposed for value prediction including last value prediction [3], stride prediction [9], context prediction [10, 11], and hybrid approaches [10, 11, 12]. In our scheme, we use value prediction to predict the instruction result. We use the bitmap, like MIPS R10000's renaming mechanism [19], which index to the physical registers. When the instruction's output can be predicted, the corresponded bit will be set. By using the bitmap, we can identify whether the physical register was predicted or not through rename stage.

3.2.1 The Value Prediction Microarchitecture

Figure 3 shows the position of value prediction's operations in our simulated pipeline. Figure 4 shows the position in microarchitecture data-path.

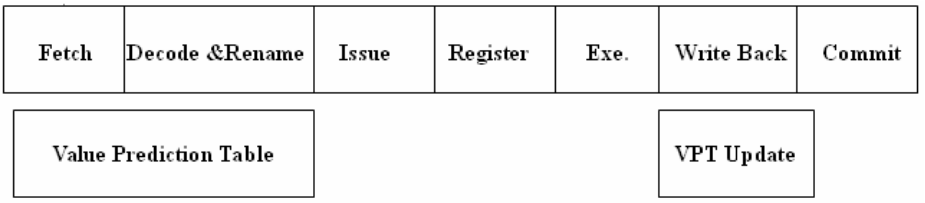


Fig. 3. Positions in the Pipeline of VP-Specific Operation

The Instruction Fetch Unit fetches and places the instructions in the Instruction Queue. At fetch stage, PC of each instruction is also used for value prediction table (VPT) access. At rename stage, the result of instruction from VPT can be stored in the instruction's corresponding physical register. At the same time the register corresponding bit in the bitmap will be set. When instructions arrive to the write back stage, the results of executions were be used to update the VPT. Other instructions which need the predicted instruction's result will check the bitmap to decide the instructions to speculative wakeup or not at the rename stage. The validation logic will verify the prediction result and send a signal to reissue the dependence instructions.

3.3 Issue Logic Design

In this section, we will describe the proposed architecture. We will explain how to design the issue logic and how to operate in superscalar in detail. At first, we will introduce the value prediction microarchitecture in our experiment. Then, we will introduce how to operate with RUU and speculative wakeup logic. At last, we will discuss the recovery mechanism.

3.3.1 Issue Logic

From figure 2 we knew that instructions spend most time on waiting for the last operand. Therefore we proposed the new issue logic in figure 5. In our issue logic only the last operand will adopt the prediction result. We can speculatively wake up instructions which have one ready operand and one predicted operand. At issue stage, if the predicted operand becomes ready the speculative issue mechanism will be disabled. Therefore, it will not need the recovery when the prediction is proven to be incorrect.

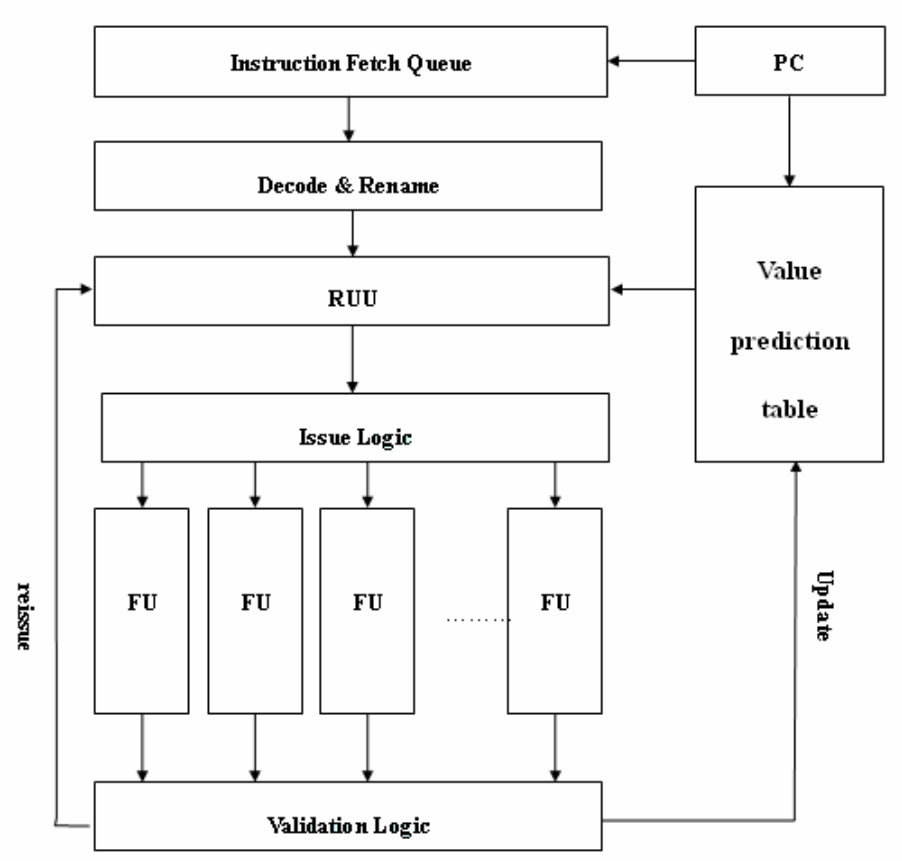


Fig. 4. Data-Path with VPT

In figure 5 the SPEC RPR and SPEC LPR bit set at rename stage, check the bitmap of value prediction result, it didn't need the extra comparator. Not like in [6, 14] speculative wakeup mechanism need add the comparator to check the grandparents ready. The other speculative wakeup (as figure 1) may be having operand not ready.

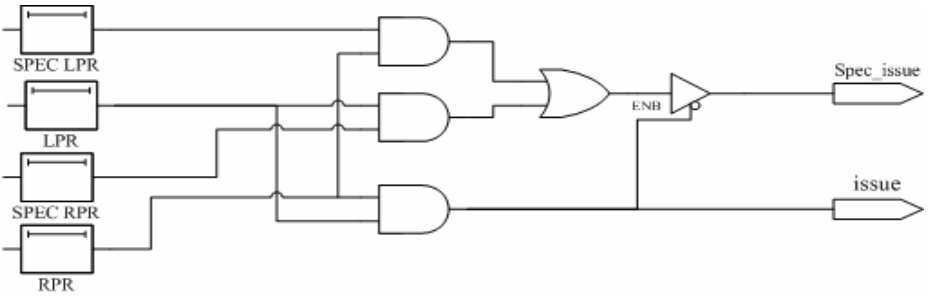


Fig. 5. The Issue Logic

In order to keep some information about the value prediction, we add four fields in RUU as figure 6 shown in. The first field is a bit about execution states, it is set when the instruction is speculative issue to the function unit. The instructions depend on the prediction result must inherit the state. Until the prediction result was verified the field will be reset. The second (third) field is a bit about the operand speculative ready or not. When instruction goes through the rename stage, the issue logic will check the bitmap to see if the value prediction result is ready or not. When the value prediction result is produced, the field will be set. If one of operand is ready and another operand is predicted, instruction is able to speculative wakeup. The last field is a 2-bit saturating counter, it use to decide issue priority. Each waked up instruction has different initial priority value. After instruction was waked up, the priority counters will incremented each cycle. If instruction result was predicted, the priority value was 3. If instruction was speculative waked up it was set to 0. The values of instructions are set to 1. The select logic picks the instructions with the highest priority value to execute.

VP_spec_mode	Spec_Lop_ready	Spec_Rop_ready	Priority_value_counter

Fig. 6. The RUU Field

VP_spec_mode: was instruction speculatively issue
Spec_Lop_ready: was the left operand predicted
Spec_Rop_ready: was the right operand predicted
Priority_value_counter: the issue priority

3.3.2 Recovery Mechanism

When result of an instruction is predicted, the issue logic will allocate a vector to record the speculatively issue instructions. The vector's width is set to the size of RUU. If instruction was used the predicted value of operand speculatively issue, the corresponding bit of the vector must be set. When the value prediction was incorrect, the instructions that corresponding bits was set on the vector will be reissued. If the value prediction was correct, the corresponded vector will be released that means no recovery is needed.

4 Evaluation Methodology

We describe our simulation environment in this section. First, we explain the machine model used in our simulation study. Then, we describe the simulator set up and the benchmark programs used.

4.1 Machine Model

The simulators used in this work are derived form the SimpleScalar 3.0 tool set [1], a suite of functional and timing simulation tools. This architecture is based on the Register Update Unit (RUU), which is a mechanism that combines the instruction window for instruction issue, the rename logic, and the reorder buffer for instruction commit under the same structure. The instruction set architecture employed is the Alpha instruction set, which is based on the Alpha AXP ISA. Two kinds of branch predictor—hybrid, and perfect—are used for performance evaluation. Table 1 summarizes some of the parameters used in our baseline architecture. Table 2 presents the parameters of value predictor.

Table 1. Machine Configuration for Baseline Architecture

Instruction fetch	4 lines per cycle. Only one taken branch per cycle. IFQ size is 32
Branch predictor	hybrid: gshare + bimodal (default) 64 entries BTB. 3 cycles mispredict penalty
Out-of-Order execution mechanism	Issue of 4 instructions/cycle 128 entries RUU (which is the ROB and the IW combined) 32 entry load/store queue. Loads executed only after all preceding store addresses are known. Value bypassed to loads from matching stores ahead in the load/store queue.
Functional units (FU)	4-integer ALUs 2 load/store units 2-FP adders, 1-Integer MULT/DIV, 1-FP MULT/DIV
FU latency	int alu--1, load/store—1, int mult--3, int div--20, fp adder--2, fp mult--4, fp div—12, fp sqrt--24
L1 D & I cache	64K bytes, 2-way set assoc., 32 byte line, 1 cycles hit latency. Dual ported.
L2 D & I cache	256K bytes, 4-way set assoc., 64 byte line, 12 cycles hit latency
Memory	Memory access latency (first-18, rest-2) cycle. Width of memory bus is 32 bytes.
TLB miss	30 cycles

Table 2. Value Predictor Configuration

	Configuration
Last	4K VHT (Value History Table)1K classification table, 2bit counter
Stride (default)	4K VHT
2 level	4K VHT, 4K PHT (Pattern History Table)4 history data value, Threshold = 3
Hybrid	Stride + 2lev (Threshold = 6)

4.2 Evaluation Method

We compare performance for the following configurations:

Table 3. Evaluation Configuration

Baseline	Baseline architecture
Sepc_issue_logic	Issue logic proposed in this paper

4.2.1 The Benchmark and Input Set

To perform our experimental study, we have collected results for the integer SPEC2000 benchmarks. The programs were compiled with the *gcc* compiler include in the tool set. Table 4 shows the input data set for each benchmark. In simulating the benchmarks, we skipped the first billion instructions, and collected statistics on the next fifty million instructions.

Table 4. Input set for benchmark

SPECInt' 2000	Input
bzip2	input.source
crafty	crafty.in
gap	ref.in
gcc	166.i
gzip	input.graphic
mcf	inp.in
parser	ref.in
twolf	./twolf/ref
vortex	lendian1.raw
vpr	net.in, arch.in

5 Performance Analysis

In this section, we present the simulation results of our experiment. We will examine the performance improvement gained by using the proposed mechanism. Then we will compare the performance of the branch, issue width, and different RUU sizes.

5.1 Experiment Results

Figure 7 shows the IPC of the different architectures. The average speedup over baseline architecture is 18.02%. In some benchmarks, like bzip2 and vortex, the performance improved not very obvious. The speedups are 2.64% for bzip2, 5.57% for vortex. The bzip2 having high ILP lead to the performance improved not very obvious. Instructions of vortex are very difficult to predict. Hence, IPC of vortex was low.

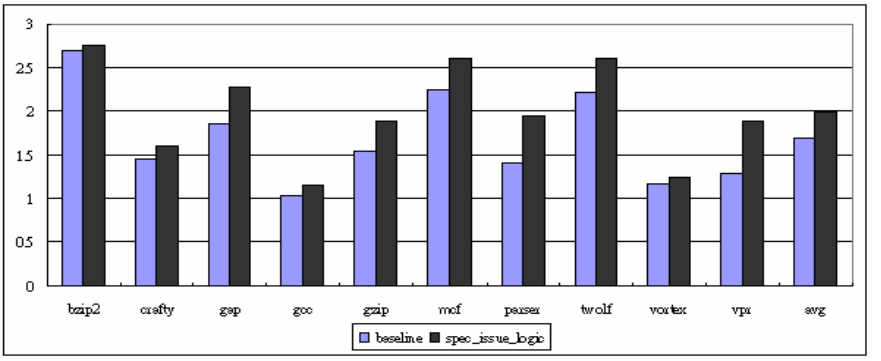


Fig. 7. IPC

5.2 The Effects of Branch Prediction

Figure 8 shows the normalized IPC with perfect branch prediction. The average speedup is 9.78% for our proposed architecture. The improved performance was not very obvious. The average baseline's IPC was 2.0195. It was like the case of bzip2 in the previous before section.

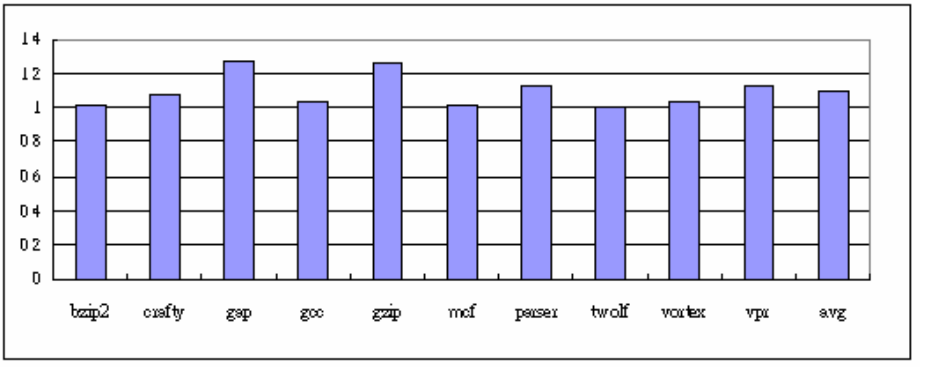


Fig. 8. Normalized IPC With Prefect Branch Prediction

5.3 Issue Width

Figure 9 shows the three machines with 4, 8, and 16 instructions width, respectively. We can see obviously that enhance the width from 4 to 8 will improve the performance in our model. This is because there are not enough instructions to wakeup in other two cases. In all models, enhance width from 8 to 16 will not improve the performance notably in SPECint2000. The main reason is that more ILP is needed in 16 bandwidth's processors. All models are not having enough instructions to execute per cycle.

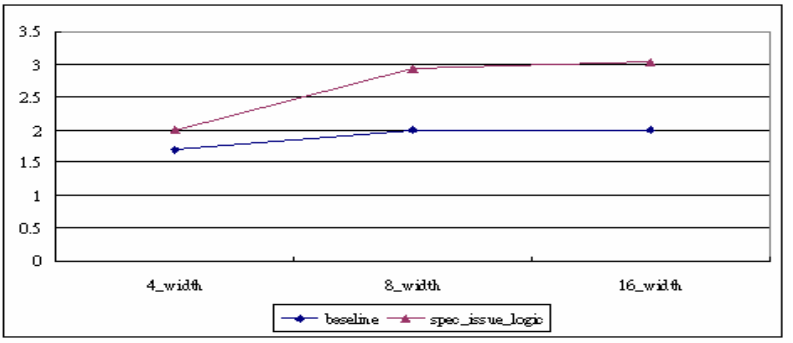


Fig. 9. IPC of Different Bandwidth

5.4 RUU Size

Figure 10 shows the normalized IPC with different RRU sizes. The IPC value were normalized to the corresponding baseline's size In average, our model has 20% speedup when the RRU sizes were set to 32.

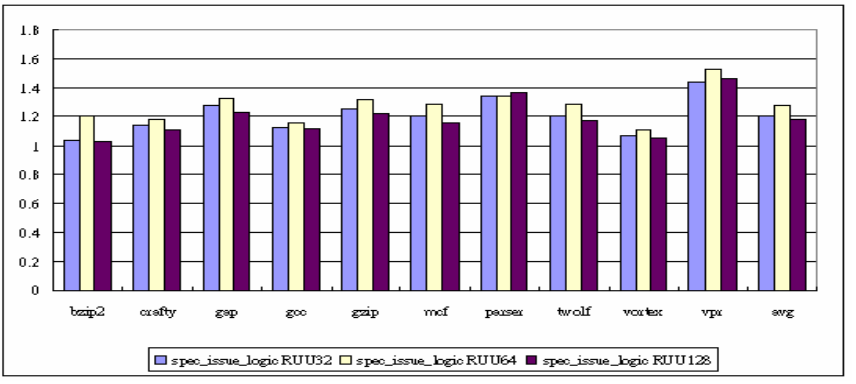


Fig. 10. Normalized IPC of Different RUU Size

6 Conclusions

In modern superscalar processors, the main part of dynamic scheduling is wakeup and select logic. Issue logic in out-of-order processors are the most important part of our research. Conventional issue logic is not very efficiency because they don't issue enough instructions to functional units. That is, there are many instructions waiting to issue. This will cost a lot of time and then impact the performance.

In this paper, we propose the speculative issue logic with value prediction mechanism. It can wakeup more instructions to execute and exploit the instructions parallelism. We also propose the priority value of the select logic. In our experiment, our model will enhance performance by 18.02%.

References

1. D. Burger and T. M. Austin. “The SimpleScalar tool set. Technical Report TR-97-1342”, University of Wisconsin, 1997.
2. Sang-Jeong Lee; Pen-Chung Yew “On augmenting trace cache for high-bandwidth value prediction” Proceedings of the IEEE Transaction on Computers, Sept. 2002
3. M. Lipasti, and J. Shen. “Exceeding the Dataflow Limit via Value Prediction.” In Proceedings of the 29th International Symposium on Microarchitecture (MICRO-29), Dec 1996.
4. Gurindar S. Sohi, “Instruction Issue Logic for High-performance, Interruptible, Multiple Functional Unit, “ Pipelined Computers, IEEE Transaction on Computers, March 1990.
5. Chio-Hung Liao and Jong-Jiann Shieh, ”Exploiting Speculative Value Reuse Using Value Prediction” Melbourne, Victoria, Australia, ACSAC`2002, Jan. 2002
6. Feng-Jiann Shiao and Jong-Jiann Shieh ,“An Issue Logic for Superscalar Microprocessors”, 16th International Conference on Computer Applications in Industry and Engineering, Las Vegas, Nevada, USA, CAINE-2003, Nov. 2003
7. Brown, M.D.; Stark, J.; Patt, Y.N.; “ Select-free Instruction Scheduling Logic,“ In Proceedings of the 34th International Annual ACM/ IEEE on Microarchitecture (MICRO-34) ,2001.
8. Mehis, A.; Radhakrishnan, R.; “Optimizing applications for performance on the Pentium 4 architecture” In Proceedings of the 2002 IEEE International Workshop on Workload Characterization, 25 Nov. 2002
9. F Gabbay and A Mendelson. “Using Value Prediction to Increase the Power of Speculative Execution Hardware” ACM Transactions on Computer Systems, Vol. 16, No. 3, August 1998 .
10. Y. Sazeides, and J. Smith. The Predictability of Data Values. In Proceedings of the 30th International Symposium on Microarchitecture (MICRO-30), Dec. 1997.
11. K. Wang, M. Franklin. “Highly Accurate Data Value Predictions Using Hybrid Predictor.” In Proceedings of the 30th International Symposium on Microarchitecture (MICRO-30), Dec. 1997.
12. B. Rychlik, J. Faistl, B. Krug, A. Kurland, J. Jung, Miroslav, N. Velev, and J. Shen. “Efficient and Accurate Value Prediction Using Dynamic Classification.” Technical Report of Microarchitecture Research Team in Dept. of Electrical and Computer Engineering, Carnegie Mellon Univ., 1998.
13. B. Rychlik, J. Faistl, B. Krug, and J. Shen. “Efficacy and Performance Impact of Value Prediction.” Parallel Architectures and Compilation Techniques, Paris, Oct. 1998.

14. Stark, J.; Brown, M.D.; Patt, Y.N.; "On pipelining dynamic instruction scheduling logic" In Proceedings of the International Symposium on 33rd Annual IEEE/ACM ,Dec. 2000 10-13
15. Ernst, D.; Hamel, A.; Austin, T.; "Cyclone: A Broadcast-Free Dynamic Instruction Scheduler with Selective Replay" In Proceedings of the 30th Annual International Symposium on Computer Architecture, June 2003.
16. Ernst, D.; Austin, T.; "Efficient Dynamic Scheduling Through Tag Elimination" 29th Annual International Symposium on Computer Architecture, May 2002.
17. Fields, B.; Rubin, S.; Bodik, R.; "Focusing processor policies via critical-path prediction" 28th Annual International Symposium on Computer Architecture, July 2001
18. Chao-ying Fu; Bodine, J.T.; Conte, T.M.; "Modeling value speculation: an optimal edge selection problem" In Proceedings of the IEEE Transactions on Computers, March 2003
19. K.C. Yeager. The MIPS R10000 Superscalar Microprocessor, IEEE Micro, April 1996.

Using Decision Trees to Improve Program-Based and Profile-Based Static Branch Prediction

Veerle Desmet, Lieven Eeckhout, and Koen De Bosschere

Department of Electronics and Information Systems,
member HiPEAC, Ghent University—UGent, Belgium
{veerle.desmet, lieven.eeckhout, koen.debosschere}@elis.UGent.be

Abstract. Improving static branch prediction accuracy is an important problem with various interesting applications. First, several compiler optimizations such as code layout, scheduling, predication, etc. rely on accurate static branch prediction. Second, branches that are statically accurately predictable can be removed from the dynamic branch predictor thereby reducing aliasing. Third, for embedded microprocessors which lack dynamic branch prediction, static branch prediction is the only alternative.

This paper builds on previous work done on evidence-based static branch prediction which uses decision trees to classify branches. We demonstrate how decision trees can be used to improve the Ball and Larus heuristics by optimizing the sequence of applying the heuristics and by discovering two new heuristics, namely one based on the post-domination relationship between the current basic block and its successor and one based on the dependency distance between the branch and its operand defining instruction. Experimental results indicate an increase in the number of instructions per mispredicted branch by 18.5% on average for SPECint95 and SPECint2000. In addition, we show that decision trees can improve profile-based static branch prediction by up to 11.7% by predicting branches that are unseen in the profile runs.

1 Introduction

Static branch prediction is an important research topic for several reasons. Compilers rely on accurate static branch prediction for applying various compiler optimizations, such as code layout, instruction scheduling, register allocation, function inlining, predication, etc. Moreover, in many cases the applicability or the effectiveness of the compiler optimizations is directly proportional to the branch prediction accuracy. Second, branches that are highly predictable using static branch prediction or hard to predict dynamically can be excluded from the dynamic branch predictor thereby reducing aliasing in the predictor and thus increasing the predictor's performance [9]. The IA-64 ISA for example, provides branch hints to communicate to the hardware whether the branch should be updated in the dynamic branch predictors. Third, several embedded processors lack a dynamic branch predictor, e.g. the Philips' TriMedia and the TI VLIW family processors. For these microprocessors, static branch prediction is the only source for reducing the number of branch mispredictions.

T. Srikanthan et al. (Eds.): ACSAC 2005, LNCS 3740, pp. 336–352, 2005.

1.1 Background

There exist two approaches to static branch prediction, namely *program-based* and *profile-based* branch prediction. The first approach only uses structural information of a computer program. A well known example of program-based branch prediction are the Ball and Larus heuristics. Ball and Larus [1] present a set of heuristics that are based on the branch opcode, the branch operands, and properties of the basic block successors of a branch. Ball and Larus propose to apply these heuristics in a fixed ordering which means that the first heuristic that applies to a particular branch will be used to predict the branch direction. To determine the best ordering they evaluate all possible combinations. Instead of using a fixed ordering for applying the Ball and Larus heuristics, Wu and Larus [17] propose using the Dempster-Shafer theory for combining heuristics in case multiple heuristics apply to a particular branch. The results in [3] however, indicate that this does not improve the branch prediction accuracy.

A second example of program-based branch prediction is evidence-based static branch prediction (ESP) proposed by Calder *et al.* [3]. In evidence-based branch prediction, machine learning techniques are used to classify branches based on a large number of branch features. We will use decision trees for classification since they are equally performing as neural networks while being easier to interpret [3]. The biggest advantage of this approach is that decision trees are generated automatically so that they can be specialized for a specific programming language, a specific compiler, a specific ISA, etc. Other program-based techniques rely on heuristics that are based on intuition and empirical studies and thus are not easily transformed to another environment.

The profile-based branch prediction approach uses information obtained from previous runs of the same program with different inputs. According to Fisher and Freudenberger [7], branches go in one direction most of the time; as such, one can predict the direction of a branch well using previous runs of the same program. Profile-based static branch prediction is widely recognized as being more accurate than program-based prediction [3, 7]. However, profile-based static branch prediction has several disadvantages. First, gathering profile data is a time-consuming process that programmers are not always willing to do. Second, profiling is not always practical, e.g. for an operating system kernel, or even infeasible for real-time applications. Third, the selection of representative inputs might be difficult.

In this paper, we consider both static branch prediction approaches since both come with their advantages. Program-based branch prediction has a lower cost and profile-based branch prediction has a higher accuracy.

1.2 Contributions and Outline

This paper makes the following contributions. First, we demonstrate that decision trees can be used to improve the Ball and Larus heuristics by automatically optimizing the sequence of applying these heuristics. Second, we increase the number of instructions per mispredicted branch by 18.5% over the Ball

and Larus heuristics by proposing two new heuristics, one based on the post-dominator relationship of the current block and its successor and one based on the dependency distance between the branch and its operand defining instruction. Third, we show that decision trees can also be used to improve the accuracy of profile-based static branch prediction by up to 11.7%. In particular, for branches that are unseen in the profile runs, we propose using decision trees. The experimental results are obtained using the SPECint95 and SPECint2000 benchmarks.

This paper is organized as follows. After detailing the experimental setup in section 2, we will discuss in section 3 how evidence-based branch prediction can be improved by adding new static branch features. In section 4, we discuss the Ball and Larus heuristics and show how decision trees can be used to improve their performance. In section 5 we demonstrate that profile-based branch prediction can also benefit from decision trees when used in conjunction. Finally, we discuss related work in section 6 and conclude in section 7.

2 Experimental Setup

The experimental results in this paper are obtained for the SPECint95 and the SPECint2000 benchmarks. We did not include gcc, perl and vortex from SPECint95 because our evaluation methodology uses cross-validation which

Table 1. Benchmarks, their inputs and some branch statistics. 'Stat.' is the number of static branches executed; 'Dyn.' is the number of dynamic branches executed in millions; 'Best' is the upper limit for IPM for static branch prediction. Inputs with an asterix* are only used in Section 5 and are excluded from the average.

Program	Input	Stat.	Dyn.	Best	Program	Input	Stat.	Dyn.	Best
gzip	graphic*	830	7,939	102.14	vpr	place*	1,584	292	75.29
	log*	833	3,861	129.01		route	2,463	7,671	128.59
	program*	832	8,031	129.39	mcf	ref	844	10,758	50.08
	random	777	6,748	241.05	crafty	ref	2,677	15,990	67.93
	source*	851	9,351	104.94	gap	ref	4,757	21,975	169.42
gcc	166*	18,588	4,136	358.23	vortex	lendian1*	5,930	12,966	1037.62
	200*	17,581	12,011	122.58		lendian2	5,940	13,779	730.65
	expr*	17,904	1,243	146.16		lendian3*	5,923	14,500	1092.65
	integrate*	16,720	1,281	229.12	bzip2	graphic*	854	12,697	140.25
	scilab	17,771	6,516	113.30		program	854	11,612	106.70
parser	ref	2,624	60,851	64.02		source*	853	9,862	102.14
perlbmk	splitmail 850*	6,146	12,309	440.06	twolf	ref	2,976	34,249	63.44
	splitmail 704*	6,147	6,508	402.49	compress	bigtest	426	5,004	72.66
	splitmail 535	6,146	6,072	421.05	go	5stone21	4,963	3,872	39.00
	splitmail 957*	6,148	10,822	402.55	ijpeg	penguin	1455	1,485	134.80
	makerand*	2,029	177	318.10	li	ref	918	8,532	50.98
	diffmail*	5,598	3,774	154.18	m88ksim	ctl.raw.lit	1,113	1,969	202.79
average									115.94

means that the program being evaluated is not included in the train set; including these three benchmarks would have been unfair as they appear in both SPECint95 and SPECint2000. We did not include `eon` since it is the only SPECint2000 benchmark written in C++ whereas the other benchmarks are all written in C. Indeed, Calder *et al.* [3] showed that ESP is sensitive to the programming language in which the benchmarks are written and that therefore separate decision trees should be considered for different programming languages. All the benchmarks are compiled with the Compaq C compiler version V6.3-025 with optimization flags `-arch ev6 -fast -O4`. An overview of the benchmarks and their inputs is listed in Table 1. All the inputs are reference inputs and all the benchmarks were run to completion. In Table 1, for each benchmark-input pair the lower limit in branch misprediction rate is shown for static branch prediction. This lower limit is defined by the most likely direction for each branch. For example, for a branch that is executed 100 times of which 80 times taken, the best possible static branch prediction achieves a prediction accuracy of 80%. All the averages reported in this paper are geometric averages over all the benchmarks. Our primary metric for measuring the performance of the proposed branch prediction techniques is the number of instructions per mispredicted branch (IPM) which is a better metric than misprediction rate because IPM also captures the density of conditional branches in a program [7]. Detecting basic blocks and loops in the binaries was done using Squeeze [5], a binary rewriting tool for the Alpha architecture. Squeeze reads in statically linked binaries and builds a control flow graph from it. Computing the static branch features that will serve as input for the decision trees is done using a modified version of Squeeze.

3 Evidence-Based Prediction

This section presents a background on evidence-based branch prediction (ESP) using decision trees. We also show how ESP can be improved by stopping the growth during decision tree learning and by adding a set of newly proposed branch features.

3.1 Decision Trees

Decision trees consider static branch prediction as a classification problem: branches are classified into 'taken' and 'not-taken' classes based on a number of static branch features. A decision tree consists of a number of internal nodes in which each node discriminates on a given branch feature, and in which the leafs represent the branch classes 'taken' and 'not-taken'. A decision tree can thus be viewed as a hierarchical step-wise decision procedure.

In our experiments, the decision trees are generated using C4.5 [11]. Developed by J. Ross Quinlan in 1992, C4.5 is a widely used tool for constructing decision trees; C4.5 was also used by Calder *et al.* [3]. Our slightly modified version of C4.5 takes into account the branches' execution frequencies giving a

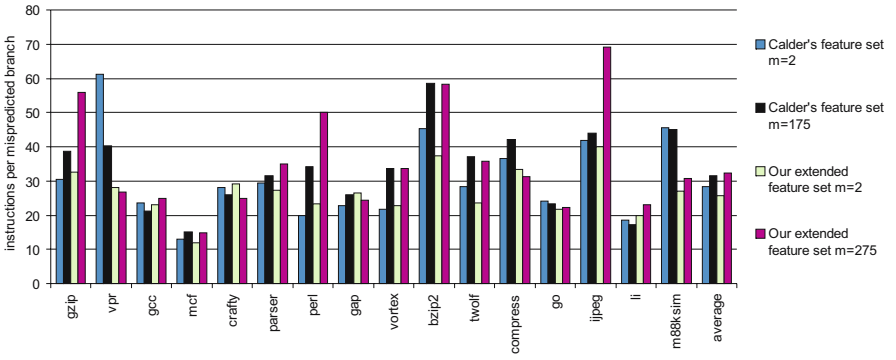


Fig. 1. Number of instructions per mispredicted branch for evidence-based branch prediction using decision trees

higher weight to more frequently executed branches; this is done by adding an extra attribute to the input of C4.5 for each static branch. To achieve the same result, Calder *et al.* [3] duplicates static branches proportional to their execution frequency. As done in [3], the execution frequencies were rescaled by multiplying the normalized frequencies by a factor 1,000; branches with a rescaled value smaller than 1 were excluded from the training set. This eliminates a number of low-frequency branches and allows C4.5 to focus on the important frequently executed branches. An additional advantage of eliminating low-frequency branches is that it seriously reduces the time to construct the decision trees. Further, the default C4.5 training and pruning settings are used.

All the results in this paper are obtained by performing a cross-validation study, meaning that the benchmark under evaluation is excluded from the training set. Cross-validation was done to provide fair results, since the predicted program is not seen during the construction of the decision tree.

3.2 Branch Features

We have applied decision tree prediction on the SPECint95 and SPECint2000 benchmarks in a cross-validation study using the feature set from Calder *et al.* [3]. The first bar in Figure 1 show the average number of instructions between two mispredicted branches equals 28.4.

3.3 Stopping the Growth of the Decision Tree

In his book on machine learning [8], Tom Mitchell describes approaches to avoid overfitting in decision tree learning. The first group of techniques allow the tree to overfit the data, and then post-prune the tree in a second phase. Pruning is the process of consecutively removing subtrees, making them leaf nodes, and assigning them the most common decision class of the training examples. Nodes

are pruned iteratively, at each step choosing the node whose removal affects the estimated classification accuracy the least. This post-pruning approach is the default strategy for C4.5.

An alternative to post-pruning is stopping the growth of the tree before it reaches the point of overfitting the data. In C4.5, this technique can be forced by setting *parameter* m to indicate the minimal number of examples in at least two subtrees when splitting a node. The default value in C4.5 for m equals 2. By setting this m, we directly tune the generalization degree to the training data. As m increases the tree concentrates only on splits with sufficient examples in the subtrees. and thus the prediction strategy becomes more general. To determine a good m, we evaluate program 1 (P1) based on data from P2 to P16 while varying m from 25 to 1000 with a step size of 25 and we repeat this experiment for P2 to P16 (cross-validation). If we would optimize the IPM as a function of m for each benchmark, we would systematically bias our results to better performance. To prevent the latter, for a program P1 we determine the average IPM for P2 to P16 for each m, and optimize for m. This technique potentially results in 16 different m values; however, for all except one the value equals $m = 175$. In addition, we observed that m-values within the same order of magnitude do not affect the results significantly. In Figure 1, the second bar displays the IPM when applying this growth stopping mechanism using the feature set by Calder *et al.* This graph shows that IPM is increased from 28.4 to 31.4 which is an increase by 10.6%. Most programs benefit substantially from stopping the growth of the tree; however, there are a few benchmarks that benefit from a specialized tree, the most notable example being `vpr`.

3.4 Additional Branch Features

In the next set of experiments we have extended the number of static branch features with the ones given in Table 2. Half of the additional features are static properties of the branch's basic block. The other half relates to the successor basic blocks. The experimental results given in Figure 1 indicate the extended set decreases the average IPM over the original feature set for the default $m = 2$. However, when the growth stopping mechanism is enabled, the average IPM increases to 32.3 which is 2.9% larger than the original set. From a detailed analysis we observed that the 'number of incoming edges to the taken successor' from Table 2 was particularly valuable. Moreover, most (11 out of 16) programs benefit from the combination of an extended feature set and a setup that stops the growth of the tree. On average, the IPM increases by 18.5% over the previous work done by Calder *et al.* [3].

3.5 Comparing with Previous ESP Work

During this analysis of ESP, we observed that our results showed a higher average misprediction rates—38%—than the 25% reported by Calder *et al.* There are three possible reasons for this. First, we used different benchmarks than Calder *et al.* did. They used SPECint92 plus a set of Unix utilities which we

Table 2. Additional static features

Feature	Description
Register	Register number used in branch instruction
Loop level	Loop level of the branch instruction (start with zero in non-loop part, increase the number for each nested loop)
Basic Block Size	Size in number of instructions
Distance	Distance between register defining instruction and the branch that uses it
RA Register	Register number for RA
RA Distance	Distance between both register defining instructions
RB Register	Register number for RB
RB Distance	Distance between both register defining instructions
Register definition	The register in the branch instruction was defined in that basic block, or not
Pointer	The pointer heuristic is applicable
Opcode	Opcode heuristic is applicable
Incoming Edges	Number of possible ways to enter basic block
	Features of the Taken and Not Taken Successor
Basic Block Size	Size of successor basic block
Incoming Edges	Number of possible ways to enter successor basic block
Pre-return	Successor contains a return or unconditionally passes control to a return
Pre-store	Successor contains a store instruction or unconditionally passes control to a block with a store instruction
Direct call	Successor contains a call
Loop Sequence	Only applicable if loop exit edge: successor is also a loop at the same level, or not

believe are much less complex than SPECint95 and SPECint2000. For several SPEC benchmarks, Calder *et al.* report a misprediction rate around 30% (32% for `gcc`), which is comparable to our results. Next to these benchmarks, Calder's study also includes several C-benchmarks (alvinn, ear and eqntott) with extremely low miss rates pulling down the average miss rate to 25%. A second possible explanation is the fact that the lower compiler optimization level used by Calder *et al.* (`-O`) results in better predictable branch behavior than when using a higher optimization level (`-O4` in our study). During investigation of how much the branch predictability is affected by the chosen compiler optimization level we found that the use of a lower optimization level in Calder's paper indeed results in better predictable branch behavior. Less optimization makes the program structure more generic so that branches are easier to predict. The latter drops the average misprediction rate by 10% between `-O4` and `-O1`, and by 3% between `-O4` and `-O`. A third explanation could be the use of a different and more recent compiler. We do not believe the difference in reported misprediction rates between this paper and Calder's work comes from the fact that we use a (slightly) smaller number of benchmarks in our analysis, namely 16 versus 23 C

programs and 20 Fortran programs used by Calder *et al.* [1], because the number of static branches in our analysis is 2 times higher than the number of static branches by Calder *et al.*

4 Heuristic-Based Prediction

In this section, we show how decision trees improve the Ball and Larus heuristics' accuracy and usability. Before going into detail on how this is done, we first give a brief discussion on the Ball and Larus heuristics as proposed in [1].

The Ball and Larus heuristics start by classifying branches as loop and non-loop branches. A branch is a loop branch if either of its outgoing edges is an exit edge or a loop backedge. A branch then is a non-loop branch if neither of its outgoing edges is an exit or a backedge. Loop branches can be predicted very accurately by the **loop heuristic** which predicts that the edge back to the loop's head is taken and that the edge exiting the loop is not taken. In our analysis these loop branches account for 35% of the dynamic branches and for 11% of the static branches. The rest of the heuristics concern non-loop branches:

- The **pointer heuristic** will predict that pointers are mostly non-null, and that pointers are typically not equal, i.e. comparing two pointers typically results in non-equality.
- The **opcode heuristic** will predict that comparisons of integers for less than zero, or less than or equal to zero will evaluate false, and that comparisons for greater than zero, or greater than or equal to zero will be true. This heuristic is based on the notion that many programs use negative numbers to denote error values.
- The **guard heuristic** applies if the register used in the branch instruction is used in the successor basic block before being defined, and the successor block does not postdominate the branch. If the heuristics applies, the successor *with* the property is predicted to be the next executed block. The intuition is that guards usually catch exceptional conditions.
- The **loop header heuristic** will predict the successor that is a loop header or pre-header and which does not postdominate the branch. This heuristic will predict that loops are executed rather than avoided.
- The **call heuristic** will predict the successor that contains a call and does not postdominate the branch as not taken. This heuristics predicts that a branch avoids executing the function call.
- The **store heuristic** will predict the successor containing a store instruction and does not postdominate the branch as not taken.
- The **return heuristic** will predict that a successor with a return will be not taken.

Coverage—measured as the number of static branches to which the heuristic applies—and misprediction rate of the individual heuristics are listed in Table 3

[1] Note that Calder *et al.* did a separate analysis for C programs and Fortran programs; he did not consider them together in one analysis.

Table 3. Coverage and misprediction rates for the Ball and Larus heuristics

Heuristic	Coverage	Misprediction rate	Perfect
Loop	35%	19%	12%
Pointer	21%	39%	9%
Opcode	9%	27%	11%
Guard	13%	37%	12%
Loop Header	26%	25%	10%
Call	22%	33%	8%
Store	25%	48%	9%
Return	22%	29%	12%

for the SPECint95 and SPECint2000 benchmarks given in section 2. The measured misprediction rates correspond to those presented by Wu and Larus [17], except for the opcode heuristic where we reach a higher misprediction rate. The reason is that we use an Alpha architecture which does not allow us to implement the opcode heuristic as originally stated: conditional branches in the Alpha ISA compare a register to zero, rather than comparing two registers as is the case in the MIPS ISA (for which the Ball and Larus heuristics were developed). The opcode heuristic was implemented by applying the heuristic to the compare instruction (`cmp`) that defines the branch's operand. Calder *et al.* [3] also obtain a higher misprediction rate for the opcode heuristic for the Alpha ISA than for the MIPS ISA.

4.1 Optimal Heuristic Ordering

As already pointed out by Ball and Larus [1], the ordering of the heuristics can have an important impact on the overall misprediction rate. Ball and Larus came up with a fixed ordering for applying their heuristics, which is: *Loop* → *Pointer* → *Call* → *Opcode* → *Return* → *Store* → *LoopHeader* → *Guard*. As soon as one heuristic applies, the branch is predicted along that heuristic and all other heuristics possibly applying are ignored. If no heuristic applies a *Random* prediction is made. For the above ordering of heuristics, we measure an average IPM of 31.3, see Figure 2. The coverage for each heuristic in this ordering is 35%, 13%, 13%, 4.5%, 7%, 8%, 1.5%, 1%, respectively. This sums to a heuristic coverage of 83%, the remainder is randomly predicted.

To identify the optimal ordering, Ball and Larus evaluated all possible combinations. Note that the total number of orderings grows quickly as a function of the number of heuristics—more in particular, for n heuristics, there exist $n!$ orderings. As such, evaluating all possible combinations quickly becomes infeasible as the number of heuristics increases (as will be the case in the next subsection). In addition, determining the optimal ordering for one particular ISA, programming language, and compiler (optimization level), does not guarantee a well performing ordering under different circumstances embodying another ISA, compiler or programming language. Therefore, it is important to have an au-

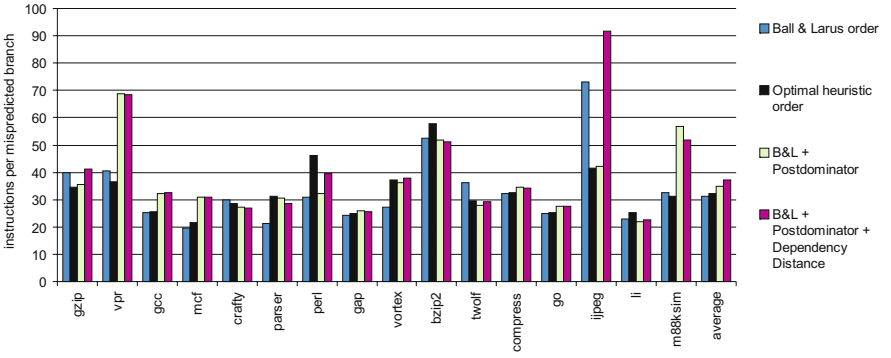


Fig. 2. The number of instructions per mispredicted branch for heuristic-based branch prediction

tomated and efficient way of finding a well performing ordering. This section proposes decision trees for this purpose.

As input during decision tree learning, we provide the evaluation of each heuristic for every static branch together with its most taken direction. We then applied our decision tree learning tool C4.5 on this data set in a cross-validation setup, i.e. for each benchmark under evaluation we build a separate tree using the information for the remaining 15 benchmarks. As such, we obtain 16 decision trees. Inspection of the trees however, revealed that most of them were quite similar to each other. The ordering obtained from this analysis is the following: $Loop \rightarrow Opcode \rightarrow Call \rightarrow Return \rightarrow LoopHeader \rightarrow Store \rightarrow Pointer$. When no heuristics can be applied it chooses the $NotTaken$ decision. The overall average IPM for this new ordering now is 32.1 which is slightly better (2.5%) than the ordering proposed by Ball and Larus. For the coverage of the heuristics we now get 35%, 5.5%, 16.5%, 9%, 4%, 8%, 4% respectively, summing up to 82%. By moving the *Pointer* heuristic to the end of the chain, the other heuristics predict a larger part more accurately, and finally the uncovered branches are mostly not taken. Note that the *Guard* heuristic, which has the lowest priority in the Ball and Larus order, is completely ignored by the decision tree. Although C4.5 is not forced to do so, the tree clearly identifies an ordering for applying the heuristics. Indeed, the decision tree could have been a 'real' tree with multiple paths from the root node to the leafs instead of a tree with one single path which is an ordering.

4.2 Adding Heuristics

Given the fact that we now have an automated way for ordering heuristics, it becomes feasible to investigate whether additional heuristics would help improving the prediction accuracy of heuristic-based static branch prediction. The question however remains to determine which heuristics should be added. To answer this

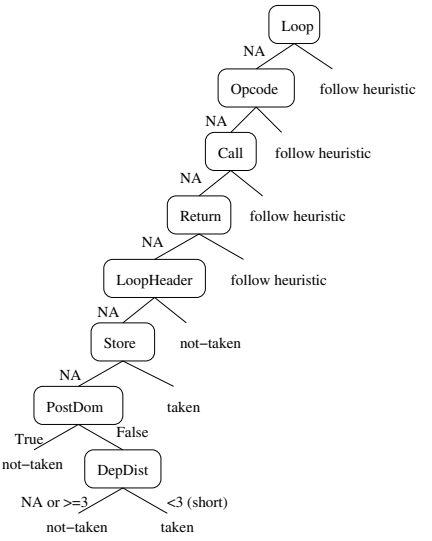


Fig. 3. Decision tree for the extended set of heuristics

question, we have done the following experiment. We have added the static features from Calder *et al.* [3] and Table 2 *one by one* to the set of Ball and Larus heuristics. Using this set of heuristics we have built up a decision tree using C4.5 and we have measured the resulting static branch prediction accuracy[2]. For each of the static features we thus have a prediction accuracy when added to the set of Ball and Larus. Subsequently, the static feature for which the extended set of heuristics achieved the highest prediction accuracy was selected for permanent inclusion in the extended set. Using this extended set we then iterate this algorithm using the remaining static features until the static prediction accuracy does no longer improve.

This experiment revealed two heuristics that when added to the set of Ball and Larus improve the overall prediction accuracy. The first one concerns the postdominator relationship between the successor and the current basic block. This heuristic states that if a branch has two successors of which one postdominates the current basic block, the successor that does not postdominate the current block should be predicted taken. The simplest example to which this *predict-non-postdominating-successor* heuristic applies is an if-statement (without else-block); the heuristic will then predict the if-block to be taken. The second heuristic is based on the dependency distance between the branch and its operand defining instruction, i.e. the number of instructions between producing a register value and consuming it by the branch. If the operand defining instruction is not part of the branch's basic block, the dependency distance is left undefined. This newly proposed *dependency distance* heuristic states that a

[2] Note that in this experiment we also use cross-validation. Further, we assume $m = 300$.

branch with an undefined dependency distance or a dependency distance larger than 3 should be predicted not-taken. The threshold on distance 3 was found empirically, but changing it to 2 or 4 does not significantly affect the results.

Figure 3 shows the decision tree provided by C4.5 for the extended set of heuristics, i.e. the set of Ball and Larus heuristics plus the predict-non-postdominating-successor and the dependency distance heuristics. This tree shows that the *Pointer* heuristic is replaced with our heuristic extensions. Replacing the *Pointer* heuristic by the postdominator heuristic results in a coverage increase of 2% together with a significant misprediction rate reduction for that specific class.

The number of instructions per mispredicted branch (IPM) is shown in Figure 2 with the extended set of heuristics. Adding the predict-non-postdominating-successor heuristic improves the IPM from 31.3 to 34.7; the dependency distance heuristic further improves the IPM to 37.1 which is an increase by 18.5% over the Ball and Larus heuristics. To conclude, the extended set of heuristics covers all branches because the remaining 16% are predicted by their dependency distance.

5 Profile-Based Prediction

As stated earlier in this paper, profile-based static branch prediction is more accurate than program-based branch prediction. There are however two important issues that need to be dealt with. The first issue is the selection and/or combination of profile inputs. The second issue is the prediction of branches that are unseen during profiling. The following two subsections show how decision trees can be used to address both issues. In these experiments we consider the test and train inputs as our profile inputs.

5.1 Comparing Decision Trees Versus Address-Based Prediction

The easiest way for selecting a profile input is by picking a randomly chosen input (in our settings, we randomly choose the test or train input) and to assign the most likely direction to each branch based on the observed behavior of the chosen profile input. The branch prediction accuracy for this approach is shown in the first column of Table 4 for all SPECint2000 benchmarks.

Previous work however has shown that if multiple profiling inputs are available, combining those can significantly improve prediction accuracy. Fisher and Freudenberger [7] propose three methods for combining multiple profiles: (i) *polling* gives each profile input one vote to predict the direction of a branch, decision by majority or taken in case of a tie; (ii) *unscaled* adds the taken (not taken) counts for the different profiles, majority direction is predicted; (iii) *scaled* also adds the taken (not taken) counts for the different profiles but scales these counts by the branch execution frequencies in each profile. The results in Table 4 clearly indicate that unscaled and scaled profile combining methods perform better than random selection. More in particular, the unscaled and scaled methods

Table 4. Number of instructions per mispredicted branch (IPM) for profile-based branch prediction: (from left to right) random, polling, unscaled, scaled, 'orig' decision trees using Calder's features [3] and 'extd' decision trees additionally using the features in 2, combining the extended decision trees with unscaled, combining the extended decision trees with scaled

benchmark	input	random	polling	unscaled	scaled	orig	extd	unscld+extd	scld+extd
gzip	graphic	94.22	31.90	93.32	93.33	53.31	61.82	93.32	93.33
	log	63.74	23.21	115.63	115.64	27.62	72.91	115.63	115.64
	program	68.83	40.11	79.97	79.97	50.89	67.93	79.97	79.96
	random	212.26	32.03	190.87	190.87	79.47	80.11	190.87	190.87
	source	44.34	21.31	97.00	97.00	26.23	72.88	97.00	97.00
vpr	place	29.49	25.96	29.49	29.49	35.50	35.98	36.76	36.76
	route	128.41	18.75	129.37	127.54	46.33	76.29	129.37	127.54
gcc	166	330.56	49.81	343.10	342.95	118.49	149.30	343.20	343.05
	200	103.41	24.64	108.07	106.78	52.48	67.32	108.33	107.03
	expr	141.88	27.17	142.57	142.55	70.57	89.41	142.57	142.55
	integrate	222.56	35.47	223.96	223.89	95.62	114.94	223.96	223.89
	scilab	104.86	25.14	106.61	106.56	57.26	71.44	106.86	106.81
mcf	ref	44.99	14.37	49.94	49.91	47.96	49.54	49.94	49.91
crafty	ref	62.19	28.02	61.71	64.89	53.05	58.09	61.72	64.90
parser	ref	61.58	17.57	63.91	63.77	50.82	62.60	63.91	63.77
perlbmk	splitmail 850	32.64	20.58	32.62	34.41	65.91	44.70	37.93	40.37
	splitmail 704	33.80	20.97	33.32	36.93	16.79	11.48	37.87	42.60
	splitmail 535	33.18	20.68	33.12	35.29	30.69	20.47	38.20	41.11
	splitmail 957	33.57	30.96	32.96	36.67	32.47	21.72	37.53	42.41
	makerand	58.83	42.56	63.63	63.63	63.16	19.82	63.63	63.63
	diffmail	39.70	23.69	42.89	43.92	34.38	22.63	42.89	43.92
gap	ref	89.65	24.37	94.11	87.43	73.02	72.74	94.05	87.39
vortex	lendian1	1016.59	16.14	1014.40	1018.95	121.13	249.08	1014.40	1018.95
	lendian2	724.35	17.80	726.31	725.93	105.97	203.37	726.31	725.93
	lendian3	1047.88	16.32	1045.09	1050.87	121.71	253.67	1045.09	1050.87
bzip2	graphic	133.99	27.83	133.43	134.71	90.73	55.98	133.43	134.71
	program	90.40	27.13	103.39	101.14	70.26	52.33	103.39	101.14
	source	63.73	28.26	80.18	78.81	54.08	45.46	80.19	78.82
twolf	ref	43.14	19.24	61.92	63.07	40.49	44.14	61.92	63.07
average		85.34	22.46	93.39	93.68	55.40	62.03	95.13	95.50

outperform polling and random selection. These four methods all assign branch predictions on a per-branch basis; we will therefore refer to them as address-based prediction techniques.

We now study how decision trees can be used to combine multiple profile inputs. For this purpose, we use the sets of static branch features from Calder and from Table 2 and build a decision tree based on the test and train inputs of the corresponding benchmark. [3] The accuracy of the decision tree is then evaluated for the available reference inputs, which were unseen during decision tree learning. The 'orig' column in Table 4 gives the IPM when using the feature set proposed by Calder *et al.* [3]; the 'extd' column gives the IPM using the extended set of static features (Calder's features and 2 together). These data illustrate that the extended set of features yields more accurate predictions than the original set by Calder *et al.* In addition, the decision trees perform better than polling. The comparison between unscaled/scaled and orig/extd also illustrate the discrepancy between program-based and profile-based prediction. An impor-

[3] For these experiments m=2 (default) for providing maximum flexibility to the decision trees in order to approximate profile-based techniques.

tant advantage decision trees have over address-based prediction techniques is that the decision trees can be interpreted so that programmers or compiler writers can learn more about their software's branching behavior. This information could be valuable for optimizing their software.

5.2 Combining Address-Based and Decision Tree Prediction

As stated earlier, an important problem with profile-based branch prediction is that it does not provide information about branches that are unseen in the profile runs. In our experiments, for seven inputs, i.e. one input for vpr and all for perlbmk, the profile runs did not cover all executed branches. For vpr–place, 14% static branches that account for 65% dynamic branches are uncovered; for perlbmk, 33% static branches accounting for 51% of the dynamic branches are uncovered. For the other benchmarks, the percentage dynamic branches that were uncovered by the profile runs was less than 1%.

We propose to use decision trees for unseen branches instead of randomly assigning a branch direction—in our experiments we assigned taken. For the other branches, that are seen in the profile runs, we use the unscaled profile combining approach by Fisher and Freudenberger [7]. Similar results for the scaled combination are shown in the rightmost column in Table 4. This approach increases IPM by 11.7% and 10.5% for vpr and perlbmk, respectively. As such, we conclude that decision trees can be successfully used in profile-based branch prediction for branches that are unseen in the profile runs.

6 Related Work

The first subsection on related work focuses on static branch prediction for which the primary goal is to guide compiler optimizations. The second subsection illustrates other applications of machine learning techniques.

6.1 Static Branch Prediction

Program-based static branch prediction. One of the simplest program-based heuristics is 'backward-taken/forward-not-taken' (BTFNT). This heuristic is based on the observation that loop branches are typically backward branches and as such are likely to be taken. Although simple, this heuristics was proven to be successful. Smith [13] discussed several static prediction strategies based on instruction opcodes. Bandyopadhyay *et al.* [2] used a table lookup using the branch opcode and operand types to determine the direction for non-loop branches. Wall [15] evaluated several heuristics for predicting basic block frequencies: the basic block loop nesting depth, a combination of the loop nesting depth and the distance to the call graph leaf, and two combinations of the loop nesting depth and the number of direct calls to the block's procedure.

Deitrich *et al.* [6] extended the Ball and Larus heuristics by incorporating source-level information available in a compiler when performing static branch

prediction. The source-level information they used concerns I/O buffering, exiting, error processing, memory allocation and printing.

Wong [16] also investigated in source-level prediction by introducing the use of names (macro, function, variable) for static branch prediction.

Patterson [10] used value range propagation which tracks value ranges of variables through a program. Branch prediction is then done by consulting the value range of the appropriate variable.

Profile-based static branch prediction. Savari and Young [12] developed a technique for combining profiles using information theory, with the notion of entropy. Although this approach attained good prediction accuracies, extending this approach to more than two profiles is not straightforward.

Young and Smith [18] proposed profile-based code transformations that exploit branch correlation to improve the accuracy of static branch prediction. If a branch exhibited a different behavior on different paths, they duplicated the code and provided different static predictions along the different paths.

6.2 Machine Learning

As in this work, Calder *et al.* [3] utilized decision trees, cfr. section 3.1. However, they did neither discuss the structure of the decision trees learned nor identified the applicability of decision trees for heuristic-based branch prediction.

Cavazos *et al.* [4] applied supervised learning techniques for inducing heuristics to predict which blocks of code would benefit from scheduling. The static features they used are the number of instructions in the block and the fraction of instructions of a certain category. They showed that rule induction can successfully use these features to determine whether to schedule or not.

Stephenson *et al.* [14] employed genetic programming to automatically search for effective heuristic priority functions in various compiler optimizations. Given a set of heuristics they tuned the priority function by evolving it over several benchmarks.

7 Conclusion

Static branch prediction is an important issue with several important applications ranging from compiler optimizations, to improving dynamic branch predictors, to improving performance of embedded microprocessors lacking a dynamic branch predictor. There are two well known static branch prediction techniques, namely program-based and profile-based branch prediction. The benefit of program-based prediction is its low cost, whereas profile-based is more accurate. This paper showed how decision trees, previously proposed in the context of evidence-based branch prediction, can be used to improve both static branch prediction approaches.

It is important to emphasize that the biggest advantage of using decision trees is the fact that they can be used to automatically generate static branch predictors. In other words, they are optimized (by construction) for a given

compiler, the given programming language in which the benchmarks are written and the given ISA. As such, we are aware of the fact that the experimental results that are obtained in this paper are sensitive to the chosen compiler, benchmarks and ISA. However, we strongly believe that the major contribution of this paper—showing how decision trees can be used to improve static branch prediction—will be applicable under different setups.

We have presented a set of static branch features that when added to the previously proposed set by Calder *et al.* also increases accuracy. We have shown that the use of decision trees improves the Ball and Larus heuristics for two reasons: (i) by automatically finding a well performing ordering for applying the heuristics, and (ii) by automatically finding additional heuristics. Our experimental results on SPECint95 and SPECint2000 show that these two contributions increase the IPM by 18.5%. The two additional heuristics identified in this paper are related to the postdomination relationship between the successor and the current basic block—the non-postdominating successor is predicted taken—and the dependency distance between the branch's operand and its defining instruction—short distances result in more likely taken branches. Finally, we have also shown that decision trees improve the accuracy of address-based techniques up to 11.7% when used in combination. Decision trees prove to be effective for branches that are unseen during profiling.

Acknowledgments

Veerle Desmet is supported by a grant from the Flemish Institute for the Promotion of the Scientific-Technological Research in the Industry (IWT). Lieven Eeckhout is a postdoctoral researcher of the Fund for Scientific Research-Flanders (FWO). This research was also funded by Ghent University and HiPEAC. We are indebted to Bruno De Bus for his support and modifications to Squeeze.

References

1. T. Ball and J. R. Larus. Branch prediction for free. In *PLDI*, pages 300–313, June 1993.
2. S. Bandyopadhyay, V. S. Begwani, and R. B. Murray. Compiling for the CRISP microprocessor. In *Proc. of the Spring 1987 COMPCON*, pages 96–100, Feb. 1987.
3. B. Calder, D. Grunwald, M. Jones, D. Lindsay, J. Martin, M. Mozer, and B. Zorn. Evidence-based static branch prediction using machine learning. *ACM Transactions on Programming Languages and Systems*, 19(1):188–222, Jan. 1997.
4. J. Cavazos and J. E. B. Moss. Inducing heuristics to decide whether to schedule. In *PLDI*, pages 183–194, June 2004.
5. S. Debray, W. Evans, R. Muth, and B. De Sutter. Compiler techniques for code compaction. *ACM ToPLaS*, 22(2):378–415, Mar. 2000.
6. B. L. Deitrich, B.-C. Cheng, and W. mei W. Hwu. Improving static branch prediction in a compiler. In *PACT*, pages 214–221, Oct. 1998.
7. J. A. Fisher and S. M. Freudenberger. Predicting conditional branch directions from previous runs of a program. In *5th ASPLOS*, pages 85–95, Oct. 1992.

8. T. M. Mitchell. *Machine Learning*. McGraw Hill, 1997.
9. H. Patil and J. Emer. Combining static and dynamic branch prediction to reduce destructive aliasing. In *HPCA*, pages 251–262, Jan. 2000.
10. J. R. C. Patterson. Accurate static branch prediction by value range propagation. In *PLDI*, pages 67–78, June 1995.
11. J. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, 1993.
12. S. Savari and C. Young. Comparing and combining profiles. *JILP*, 2, Apr. 2000.
13. J. E. Smith. A study of branch prediction strategies. In *ISCA*, pages 135–148, May 1981.
14. M. Stephenson, S. Amarasinghe, M. Martin, and U.-M. O'Reilly. Meta optimization: improving compiler heuristics with machine learning. In *PLDI*, pages 77–90, June 2003.
15. D. W. Wall. Predicting program behavior using real or estimated profiles. In *PLDI*, pages 59–70, June 1991.
16. W. F. Wong. Source level static branch prediction. *The Computer Journal*, 42(2):142–149, 1999.
17. Y. Wu and J. R. Larus. Static branch frequency and program profile analysis. In *27th MICRO*, pages 1–11, Nov. 1994.
18. C. Young and M. D. Smith. Improving the accuracy of static branch prediction using branch correlation. In *6th ASPLOS*, pages 232–241, Oct. 1994.

Arithmetic Data Value Speculation

Daniel R. Kelly and Braden J. Phillips

Centre for High Performance Integrated Technologies and Systems (CHiPTec),
School of Electrical and Electronic Engineering, The University of Adelaide,
Adelaide SA 5005, Australia
{dankelly, phillips}@eleceng.adelaide.edu.au

Abstract. Value speculation is currently widely used in processor designs to increase the overall number of instructions executed per cycle (IPC). Current methods use sophisticated prediction techniques to speculate on the outcome of branches and execute code accordingly. Speculation can be extended to the approximation of arithmetic values. As arithmetic operations are slow to complete in pipelined execution an increase in overall IPC is possible through accurate arithmetic data value speculation. This paper will focus on integer adder units for the purposes of demonstrating arithmetic data value speculation.

1 Introduction

Modern processors commonly use branch prediction to speculatively execute code. This allows the overall number of instructions per cycle (IPC) to be increased if the time saved for correct predictions outweighs the penalties for a mis–prediction. Various schemes are used for branch prediction, however, few are used for the prediction, or approximation, of arithmetic values.

The adder is the basic functional unit in computer arithmetic. Adder structures are used in signed addition and subtraction, as well as floating point multiplication and division operations. Hence, improved adder design can be applied to improving all basic forms of computer arithmetic. Pipeline latency is often restricted by the relatively large delay of arithmetic units. Therefore, a decrease in the delay associated with arithmetic operations will promote an increase in IPC.

Data value speculation schemes, as opposed to value prediction schemes, have been proposed in the past for superscalar processors, such as the use of stride based predictors for the calculation of memory locations [1]. Stride based predictors assume a constant offset from a base memory location, and use a constant *stride-length* to iteratively access elements in a data array.

In this paper we will discuss two basic designs for arithmetic approximation units, and use a ripple carry adder as an example. We will then investigate the theoretical performance of such an adder by the use of SPEC benchmarks. Finally we will briefly discuss modifications to a basic MIPS architecture model to employ arithmetic data value speculation.

T. Srikanthan et al. (Eds.): ACSAC 2005, LNCS 3740, pp. 353–366, 2005.

2 Approximate Arithmetic Units

2.1 Overview

Arithmetic units can provide an approximate result if the design of a basic arithmetic unit is modified so that an incomplete result can be provided earlier than the normal delay. This can be done by specially designed approximate hardware, or by modifying regular arithmetic units to provide a partial result earlier than the worst–case delay. The intermediate (approximate) result can then be forwarded to other units for speculative execution before the exact result is known. The speculative result can be checked against the exact result, provided either by a worst–case arithmetic unit, or against the full result after the worst–case delay. A comparison operation simultaneously provides the outcome of the speculation, and the exact result in the case of spurious speculation.

An approximate arithmetic unit is incomplete in some way. An approximate unit can be logically changed to provide results earlier than the worst–case completion time. In the common case, the result will match the exact calculation, and will be erroneous in the uncommon case. Such a unit will be called a "logically incomplete unit". It is also possible to provide an approximate result by overclocking the arithmetic hardware to take advantage of short propagation times in the average case. These units are called "temporally incomplete units".

It is important to test new designs under normal operating conditions to investigate actual performance. In the execution of typical programs, the assumption of uniform random input is not true. Li has empirically demonstrated that adder inputs are highly data dependent, and exhibit different carry length distributions depending on the data [2]. For instance, operands for loop counters in programs are usually small and positive, producing small carry lengths in adders. On the other hand, integer addressing calculations can produce quite long carry lengths. This observation lead Koes et al. to design an application specific adder based upon an asynchronous dynamic Brent-Kung adder [3, 4]. The new adder adds early termination logic to the original design.

2.2 Logically Incomplete Arithmetic Units

Liu and Lu proposes a simple ripple carry adder (we will call it a "Liu-Lu adder"), in which logic is structured such that it restricts the length that a carry can propagate [5]. This is, therefore, an example of a logically incomplete adder. Figure 1 shows an example of an 8-bit adder with 3-bit carry segments. The Liu-Lu adder exploits Burks et al's famous upper bound on the expected-worst-case-carry length [6]. Assuming the addition of uniform random binary data, the expected-worst-case-carry length is shown in (1), where C_N is the *expected* length of the longest carry length in the N-bit addition.

$$C_N \leq log_2(N) \ . \tag{1}$$

This upper bound was reduced by Briley to (2) in 1973 [7].

$$C_N \leq log_2(N) - 1/2 \ . \tag{2}$$

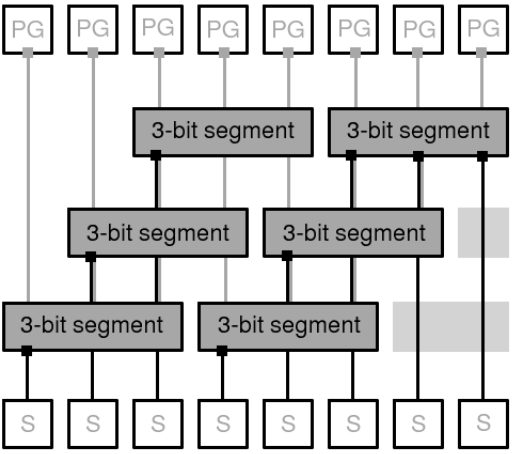


Fig. 1. Structurally incomplete 8-bit adder with 3-bit carry segments

For the purposes of adder design, we wish to find a model of the probability distribution for the expected-worst-case-carry length k-bits, in an N-bit addition, i.e.,

$$P(N,k) = 1 - Pr[C_N \geq k] = Pr[C_N < k] \; . \tag{3}$$

Lu provides an analysis of the probability of an incorrect result in the Liu-Lu adder for uniform random inputs [5]. The Liu-Lu adder forms the sum for each bit i by considering any carry generated in the previous k-bit segment (inclusive). Each k-bit segment is a ripple carry adder. (1) and (2) above show that the expected-worst-case-carry length is short with respect to the length of the addition. Hence we can tradeoff a small accuracy penalty for a significant time saving compared with the worst–case propagation delay for a ripple carry adder.

In order to predict the average–case accuracy of the Liu-Lu adder, a result is derived for the probability of a correct result, P, in an N-bit addition with a k-bit segment. Liu and Lu's equation is given below [8].

$$P_{Liu\text{-}Lu}(N,k) = \left(1 - \frac{1}{2^{(k+2)}}\right)^{(N-k-1)} \; . \tag{4}$$

Pippenger's Equation. Pippenger also provides an approximate distribution for the expected-worst-case-carry length shown in (5) below [10]. Pippenger proves this to be a lower bound to the probability distribution, and is here used as a pessimistic approximation to the performance of the Liu-Lu adder.

$$P_{Pippenger}(N,k) = e^{-N/2^{k+1}} \; . \tag{5}$$

Analysis of the Liu-Lu Equation. In the derivation of (4) Lu states that *"if we only consider k previous bits to generate the carry, the result will be wrong if the carry propagation chain is greater than* $(k+1)$*"* and *"... moreover, the previous bit must be in the carry generate condition"* [5]. Both statements are incorrect.

In analysis of arithmetic circuits, it is useful to define the result of an N-bit addition as the product of generate g_i, propagate p_i, and annihilate a_i signals for each digit $i = 0 \ldots (N\text{-}1)$ in the addition (where $i = 0$ for the least significant digit) [9].

If we consider any k-bit segment in an N-bit addition, in the Liu-Lu adder a carry will not be propagated from the k-th bit to any other bit. Hence the result in the $(k+1)$-th bit will be wrong. Therefore the Liu-Lu adder can only provide correct answers for a carry length *less than* k-bits. The approximate result will be wrong if any carry propagation chain is greater than or equal to k-bits.

Now, consider a very long carry string of length $2k$, $(g_i p_{i+1} \ldots p_{i+2k-1})$. As demonstrated above, the most–significant k-bits in the $2k$-bit segment will be incorrect, as they will not have the a carry propagated to them. Thus it is possible that an incorrect result can be produced without requiring that the previous bit to the most significant k-bit segment is in the carry *generate* condition (but it may propagate a carry generated earlier and still cause failure of the adder). Also, two *or more* disjoint carry lengths in N-bits can produce a spurious result if $k \leq N/2$.

The probability of any input being a generate signal is $P(g_i) = 1/4 = 1/2^2$, and for a propagate signal $P(p_i) = 1/2$, because it can occur in two distinct ways. Hence the probability of each k-bit segment producing an erroneous result is actually $1/2^{k+1}$. There are also $(N-k+1)$ overlapping k-bit segments required to construct the N-bit logically incomplete adder.

The result in equation 4 is arrived at by Liu and Lu's assumption *"the probability of (each k-bit segment) being correct is one minus the probability of being wrong ... we multiply all the probabilities to produce the final product"*. As discussed above, the Liu-Lu adder will produce spurious results if there exists any carry lengths $\geq k$-bits in the N-bit addition. Hence, there are many probability cross terms not represented in equation 4. The probabilities of each k-bit segment producing a carry out is fiendishly difficult to calculate as each k-bit segment overlaps with $(k-1)$ to $2(k-1)$ other such segments.

Backcount Algorithm. In order to analyse the performance of the Liu-Lu adder it is necessary to know the actual probabilities of success $P(N,k)$ for each word length N and carry segment k. Exhaustive calculation is not feasible for large word lengths, as there are 4^N distinct input combinations for a two operand N-bit adder. Set theory and probabilistic calculation are difficult due to the overlapping nature of the k-bit segments. For these reasons, a counting algorithm was devised to quickly count all the patterns of carry segments that would violate the Liu-Lu adder.

For any N and k the number of carry strings which cause the failure of the Liu-Lu adder out of 4^N possible inputs is counted. A carry string of length k-bits or more will cause the adder to fail. The algorithm works efficiently if the result for $P(N,k+1)$ is already known, as these combinations can be discounted when

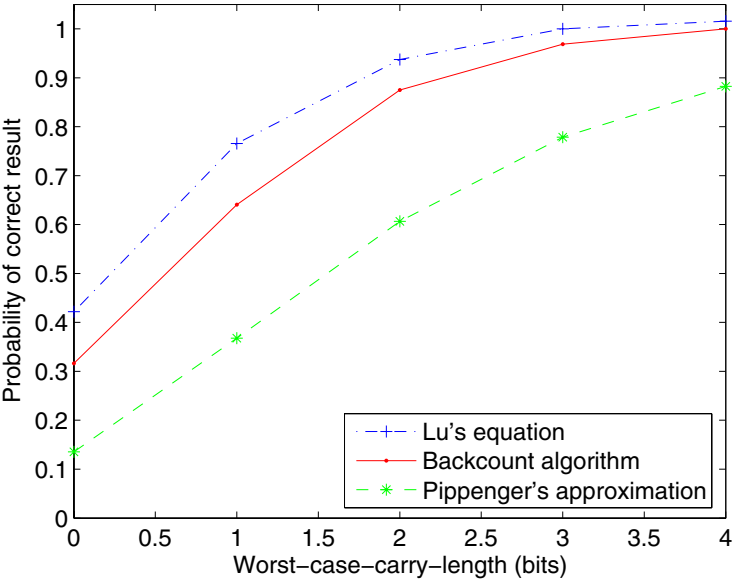


Fig. 2. 4-bit approximate adder performance

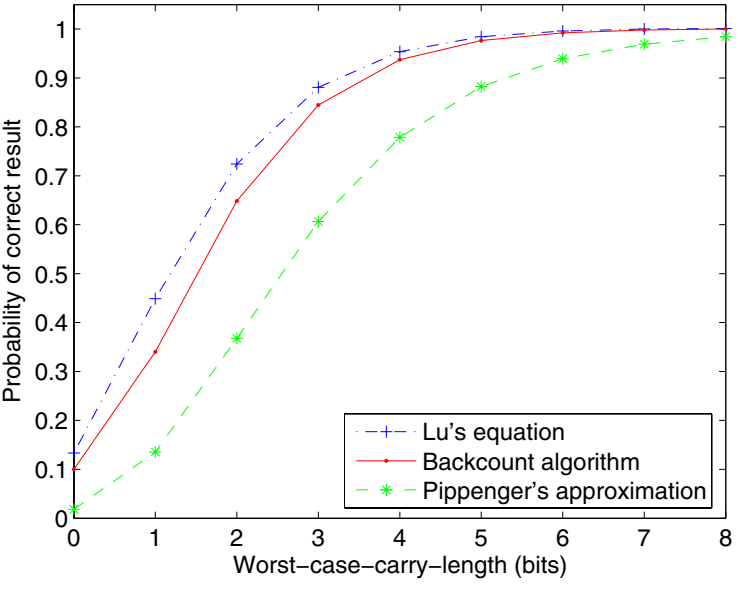


Fig. 3. 8-bit approximate adder performance

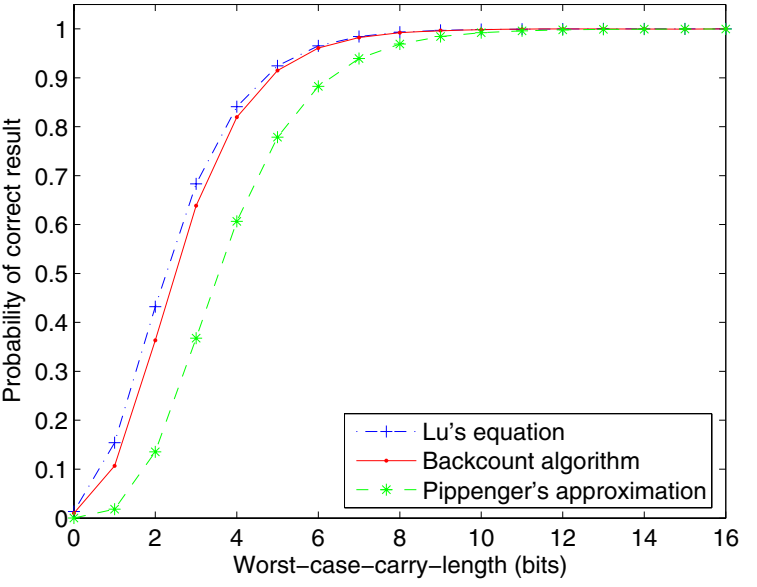


Fig. 4. 16-bit approximate adder performance

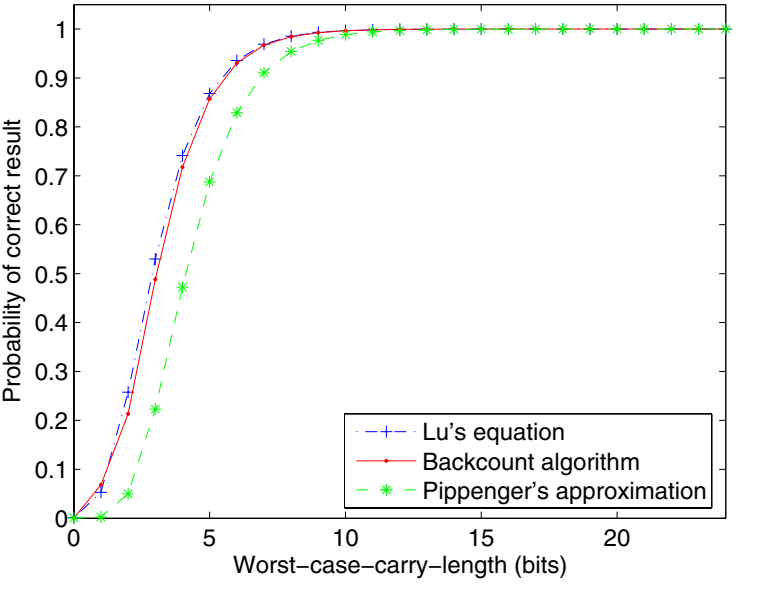


Fig. 5. 24-bit approximate adder performance

counting the violations in $P(N,k)$. For this reason we refer to the algorithm as the *backcount* algorithm.

Consider a carry string of exactly length k. The string consists of a generate signal g_i and $(k-1)$ propagate signals $p_{i+1} \ldots p_{i+k-1}$. Furthermore, the next bit, if it exists, must be an annihilate signal a_{i+k}, or the start of another carry string g_{i+k}. There are two possible ways in which a propagate signal can occur, but only one way in which a generate or annihilate signal can occur in position i. For an arbitrary k-bit segment there are r input signals (bits) to the left and s input signals to the right. So, $2^{k-1}\dot{4}^{r+s}$ possible offending combinations are counted. However, from this we must subtract all combinations containing carry lengths longer than k-bits to avoid double counting. This is achieved by recursively calling the backcount algorithm on the s- and r-bits on either side of the k-bit segment being considered, until r and s are too small.

To avoid double counting all the combinations involving multiple carry strings of length k in the N-bit addition, we must consider each case individually. This is the most time consuming part of the algorithm, as it is computationally equivalent to generating a subset of the partitions of N. The number of partitions of N increases exponentially with N, and so the process of counting many small carry chains for $k \ll N$ is very time consuming. However inefficient this may be, it has been verified to produce correct results for 8-bit addition against exhaustive calculation (see Table 1 below). The region of interest is generally $k > log_2(N)$, the expected-worst-case-carry length (Figure 1). It is inefficient for $k \leq log_2(N)$, but the calculation is reduced greatly from considering all 4^N input combinations.

A simple case exists when $k = 0$, and is included to highlight the difference in prediction against other methods. A zero-length-maximum-carry cannot exist if there are any generate signals. There are four distinct input combinations per bit and three that are not a generate signal. Hence the proportion of maximum-zero-length-carries is given below as

$$P(N,0) = \frac{4^N - 3^N}{4^N} \,. \tag{6}$$

Comparison of Models. Although the distribution given by equations 4 and 5 are not exact, it provides a sufficiently close approximation for word lengths of 32-, 64-, and 128-bits. The distribution given by equation 4 approaches the exact distribution for large N because the proportion of long carry chains to all the possible input combinations is smaller for long word lengths. However, Lu's distribution is optimistic because it does not consider all the ways in which the adder can fail. For instance, the predicted accuracy of a 64-bit adder with an 8-bit carry segment, is calculated as $P_{Liu\text{-}Lu}(64,8) = 0.9477$. Results indicate that the correct value is $P(64,8) = 0.9465$, to 4 decimal places.

Figures 2, 3, 4 and 5 show the predicted accuracy of the various methods for calculating the proportion of correctly speculated results vs. the longest carry chain in the addition. Note that to a achieve the accuracies shown with a worst-case-carry length of x-bits will require the designed adder to use $(x + 1)$-bit segments.

Table 1. Predicted number of errors for an N=8-bit logically incomplete approximate adder for various models

k-bit carry	Exact	Backcount	Lu	Pippenger
0	58975	58975	65600	64519
1	43248	43248	65536	63520
2	23040	23040	65280	61565
3	10176	10176	64516	57835
4	4096	4096	62511	51040
5	1536	1536	57720	39750
6	512	512	47460	24109
7	128	128	29412	8869
8	0	0	8748	1200

It is not a trivial task to calculate or otherwise count all possible inputs which produce carries of length k or greater. An algorithm was devised to count the number of incorrect additions of the Liu-Lu adder by considering all input combinations containing patterns of input signals consisting of g_i, p_i and a_i, which cause the N-bit adder to fail.

Further Extensions to Logically Incomplete Arithmetic Units. It is well known that in twos complement arithmetic, a subtraction operation can be performed by an adder unit by setting the carry-in bit high and inverting the subtrahend. For the case of a subtraction involving a small positive operand (or addition of a small negative operand), the operation is much more likely to produce a long carry chain. In this case it may be possible to perform the subtraction up to the expected-worst-case-carry-length, and then sign extend to the full N bits. The full advantage of this is to be determined by further investigation of distributions of worst-case-carry-lengths in benchmarked programs for subtraction operations.

2.3 Temporally Incomplete Arithmetic Units

A temporally incomplete adder is a proposed adder design that is clocked at a rate that will violate worst–case design. This adder is also based upon Burks et al.'s result (1). As the time for a result to be correctly produced at the output is dependent on the carry propagation delay, a short expected-worst-case-carry length can yield high accuracy by the same principle as the Liu-Lu adder.

The advantage of a temporally incomplete ripple carry adder is that no modifications need to be made to the worst–case design in order to produce speculative results. In order to produce exact results, the adder requires the full critical path delay, or otherwise worst–case units are required to produce the exact result.

To evaluate the performance of the temporally incomplete ripple carry adder, a VHDL implementation of the circuit was constructed using the Artisan library of components. A simulation of uniform random inputs was performed using timing information from the Artisan library datasheet. The propagation delay for a

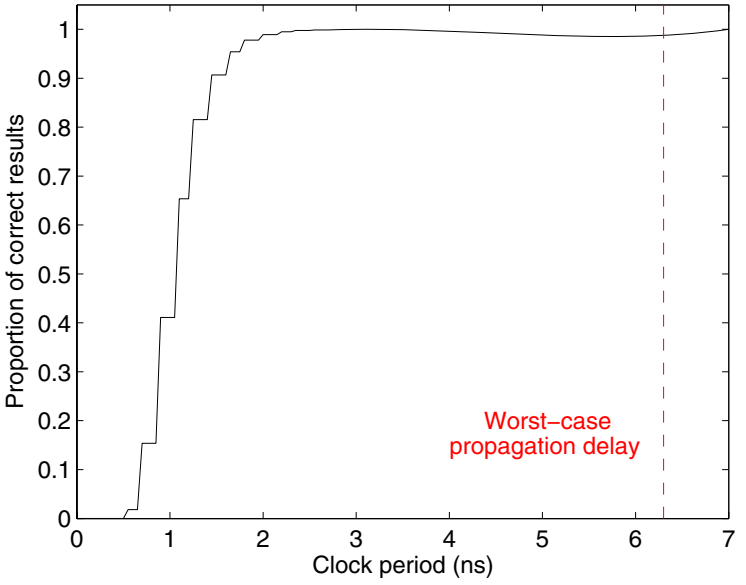


Fig. 6. Theoretical performance for a 32-bit temporally incomplete adder

full-adder cell was considered to be the average input-to-output propagation delay, due to a change in either inputs or carry–in.

The correct proportion of additions was counted when the adder was presented with 500,000 uniform random inputs. The minimum propagation time is determined by the maximum carry chain length in the addition. The number of correct additions was counted for 0.05 ns increments.

Results are shown in Figure 6. The worst case propogation delay is shown as a dashed line. Assuming uniform random input, the temporally incomplete adder can also yield high accuracy in much less than the worst–case propagation delay due to the expected short worst-case-carry-length.

Metastability. Sequential logic components like latches have setup and hold times during which the inputs must be stable. If these time restrictions are not met, the circuit may produce erroneous results. We can see from worst–case timing results above the probability of an erroneous result if the addition result arrives too late. In this case, the approximation is wrong and the pipeline will be flushed.

There is a small chance however that the circuit will become *metastable*. If the timing elements (latches or flip-flops) sample the adder output when the adder output is changing, then the timing element may be become stuck in an undetermined state. Metastable behaviour includes holding a voltage between circuit thresholds, or toggling outputs. Systems employing temporal incorrectness will, therefore, need to be designed to be robust in the presence of metastability.

3 Benchmark Investigation

3.1 Investigating Theoretical Performance

We have used the *SimpleScalar* toolset configured with the PISA architecture to simulate SPEC CINT '95 benchmarks. PISA is a MIPS-like superscalar 64-bit architecture, and supports out-of-order (OOO) execution. The pipeline has 5-stages, and supports integer and floating-point arithmetic.

To evaluate the performance of an approximate adder, we have simulated each SPEC benchmark and recorded the length of the longest carry chain. Figure 7 shows the performance of the Liu-Lu adder with a worst-case-carry-length of k-bits when used for add instructions. This figure was derived by examining all add instructions executed, irrespective of addressing mode. Note that only unsigned data was present as there were no examples of a signed add instructions in the SPEC binaries provided with *SimpleScalar*. Each benchmark in the suite is plotted on the same graph. The theoretical performance of the adder with uniform random inputs is shown as a dashed line.

We can observe that the performance of the Liu-Lu adder for add instructions is higher than expected for a small carry length, with a high proportion of correct additions achieved. However, the benchmarks repeat many identical calculations, and may be misrepresentative.

It can be observed that in some cases the length of the k-bit segment must be extended to a very wide structure in order to capture most of the additions correctly. This indicates many additions involving long carry lengths being repeated many times. Otherwise, the performance of Liu-Lu adder running SPEC CINT '95 benchmarks is close to theoretical. The Liu-Lu adder performance is also shown in figure 8 when we consider only subtraction operations. No instances of signed arithmetic were observed. All results are derived from unsigned subtractions.

When an adder is used to perform an unsigned subtraction on two binary inputs, the subtrahend is inverted, and the carry in bit is set to one. If for example we subtract 0 from 0 then one input would consist of N ones. Due to the carry in bit, the carry chain for the addition (subtraction) would propagate the entire length of the input. Likewise, subtraction operations involving small subtrahends produce large carry chains.

In subtraction the Liu-Lu adder performs much less well than theory. It is possible to approximate subtraction results by performing the full operation on k-bits out of N, and then sign–extending the result. This has the effect of reducing the calculation time for additions which result in very long carry chains. However, by sign extending past k-bits, the range of possible accurate results is reduced, as the bits of greater significance will be ignored.

Subtraction operations formed less then 2.5% of all integer arithmetic operations in the SPEC integer benchmarks. This is an important design consideration for the implementation of approximate arithmetic units. If in practice subtraction operations are not common and are not easily approximated, it is best not to use them for speculative execution.

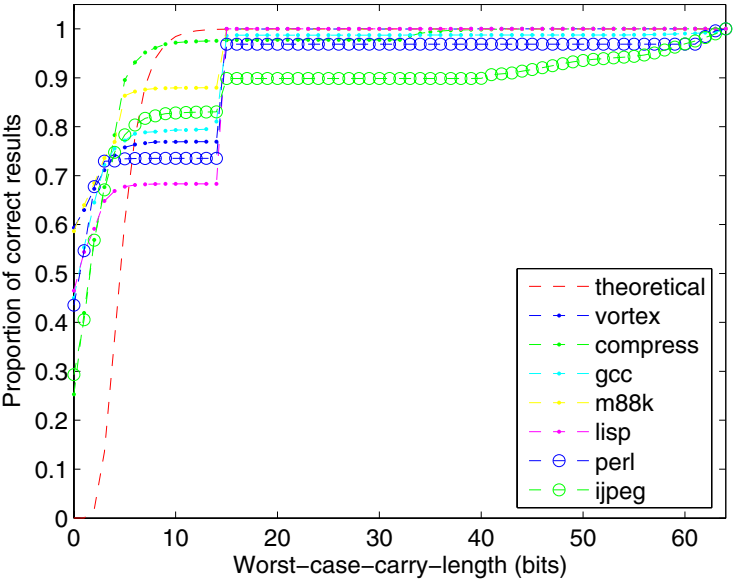


Fig. 7. Liu-Lu adder performance for ADD instructions

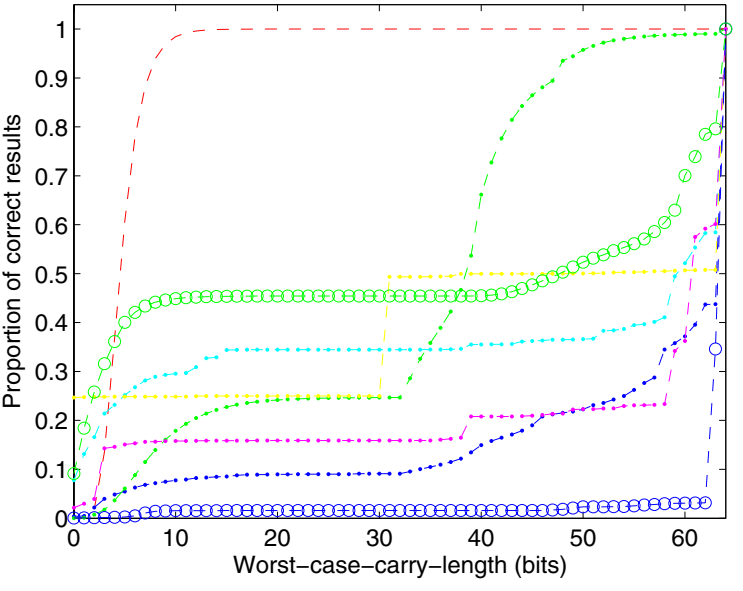


Fig. 8. Liu-Lu adder performance for SUB instructions

4 Architecture Design

4.1 Introduction

There are many challenges associated with pipelining instruction execution. Processors are becoming increasingly complex in order to exploit instruction level parallelism (ILP). Wall [11] demonstrates that in typical programs, even assuming perfect exploitation of instruction parallelism, such as a perfect branch predictors, the parallelism of most instructions will rarely exceed 5. Hence, in order to execute instructions at a higher rate assuming the maximum exploitable parallelism is fixed, instruction latency needs to be reduced (after issue).

In this section we briefly discuss the implementation of approximate arithmetic in order to facilitate speculative execution.

4.2 Pipeline Considerations

Data speculation is supported by the use of a reorder buffer (ROB). The ROB helps maintain precise exceptions by retiring instructions in order, and postponing handling exceptions until an instruction is ready to commit [12]. The use of ROB also supports dynamic scheduling and variable completion times for the various functional units.

In a branch prediction scheme, the ROB maintains instruction order. In the event that the branch outcome was different to that predicted, then it is necessary to flush all instructions after the spurious branch. This is easy to do and will not affect future execution.

In order to detect speculation dependency, extra information is needed in the ROB. Liu and Lu [8] have accomplished this by the inclusion of a value prediction field (VPF) in a MIPS architecture.

Any instruction may depend upon the results of previous instructions. If there is a dependency between instructions, and the former instruction(s) are speculative, then a store cannot be allowed to commit. If it were able to be committed, then a spurious result would be able to be written to registers or memory.

When a speculative arithmetic result is available, it is likely that a dependent instruction will start execution based upon the speculative result. In the case of a spurious arithmetic result being detected, it is necessary that all dependent instructions be flushed immediately from the pipeline and re-issued. Liu and Lu [8] make the observation that another pipeline stage needs to be added to facilitate the checking of speculative values.

As demonstrated above, different arithmetic operations have different carry characteristics, and hence suggest different approaches to data value speculation. As modern processors typically contain more than one arithmetic unit, it is easy to imagine different operations having their own dedicated hardware.

4.3 System Performance

Arithmetic data value speculation aims to improve system performance by increasing IPC. The total performance impact on a system will depend on the

programs being run, the pipeline depth, the time saved by speculation, the accuracy rate of the approximate arithmetic units and the penalty for a spurious speculation.

It is not possible to quote an absolute benefit (or detriment) to a system without a full implementation of an approximate arithmetic unit in a specific architecture, running specific programs.

In order to evaluate the performance of the approximate arithmetic units independent of the architecture, the performance of these designs has been analysed as accuracy versus time for uniform input data, and accuracy versus complexity when we consider SPEC benchmarks (simulating real programs).

System performance as raw IPC will be evaluated after a full implementation. However, before this occurs a number of architectural and engineering design problems need to be addressed, including choosing k to maximise IPC, selecting components for the designs to meet architecture specific circuit timing requirements, and increased power and area considerations.

5 Conclusion

We have demonstrated that with careful design and analysis, arithmetic approximation can quickly yield accurate results for data value speculation. Furthermore, different arithmetic operations require separate analysis in order to achieve high performance.

With continued investigation into the field of arithmetic approximation, and further research into the newly proposed concept of temporally incomplete approximate arithmetic units, data value speculation can be better implemented in specific architectures. By performing realistic performance profiling, arithmetic approximation can be better tuned to maximise the expected benefit of speculation in general computing.

References

1. Jose Gonzalez and Antonio Gonzalez. Data value speculation in superscalar processors. *Microprocessors-and-Microsystems*, 22(6):293–301, November 1998.
2. A. Li. An empirical study of the longest carry length in real programs. Master's thesis, Department of Computer Science, Princeton University, May 2002.
3. D. Koes, T. Chelcea, C. Oneyama, and S. C. Goldstein. Adding faster with application specific early termination. School of Computer Science, Carnegie Mellon University, Pittsburgh, USA, January 2005.
4. S.M. Nowick, K. Y. Yun, P. A. Beerel, and A. E. Dooply. Speculative completion for the design of high performance asynchronous dynamic adders. In *International Symposium on Advance Research in Asynchronous Circuits and Systems*, pages 210–223, Eindhoven, The Netherlands, 1997. IEEE Comput. Soc. Press.
5. S-L. Lu. Speeding up processing with approximation circuits. *Computer*, 37(3):67–73, March 2004.
6. A. W. Burks, H. H. Goldstein, and J. von Neumann. Preliminiary discussion of the design of an electronic computing instrument. Inst. Advanced Study, Princeton, N.J., June 1946.

7. B. E. Briley. Some new results on average worst case carry. *IEEE Transactions On Computers*, C–22(5):459–463, 1973.
8. T. Liu and S-L. Lu. Performance improvement with circuit level speculation. In *Proceedings of the 33rd Annual International Symposium on Microarchitecture*, pages 348–355. IEEE, 2000.
9. B. Parhami. *Computer Arithmetic: Algorithms and Hardware Designs*. Oxford University Press, New York, 2000.
10. N. Pippenger. Analysis of carry propagation in addition: an elementary approach. *Journal of Algorithms*, 42(2):317–333, 2002.
11. David W. Wall. Limits of instruction-level parallelism. *International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS*, 26(4):176 – 188, 1991.
12. Hennessey J.L. and Patterson D.A. *Computer Architecture: A Quantitative Approach*. Morgan Kauffman Publishers, San Francisco, USA, 2003.

Exploiting Thread-Level Speculative Parallelism with Software Value Prediction

Xiao-Feng Li, Chen Yang, Zhao-Hui Du, and Tin-Fook Ngai

Microprocessor Technology Labs, Intel China Research Center,
Beijing, China 100080
{xiao.feng.li, Chen.yang, zhao-hui.du,
Tin-fook.Ngai}@intel.com

Abstract. *Software value prediction* (SVP) is an effective and powerful compilation technique helping to expose thread-level speculative parallelism. With the help of value analysis and profiling, the compiler identifies critical and predictable variable values and generates speculatively parallelized programs with appropriate value prediction and misprediction recovery code. In this paper, we examine this technique in detail, describe a complete and versatile SVP framework and its detailed implementation in a thread-level speculative parallelization compiler, and present our evaluation results with Spec2000Int benchmarks. Our results not only confirm quantitatively that value prediction is essential to thread-level speculative parallelism; they also show that the corresponding value prediction can be achieved efficiently and effectively by software. We also present evaluation results of the overhead associated with software value prediction and the importance of different value predictors in speculative parallel loops in Spec2000Int benchmarks.

1 Introduction

Recent studies show value prediction is a promising technology to break the data-flow parallelism limit 5],11]. While most of the literatures on value prediction focused on *instruction-level parallelism* (ILP), our research in value prediction compilation technology shows that, value prediction is actually essential in exploring *thread-level parallelism* (TLP), and can be done purely in software with speculative multithreaded processors.

1.1 Thread-Level Speculation (TLS)

To exploit good TLP with existing programming languages, a promising approach is to use thread-level speculation (TLS), with which a piece of code (the speculative thread) runs ahead of time speculatively in parallel with the code it depends on without committing its computation results until its assumed input data are proved correct. Correctness of input data decides the delivered TLP performance.

Value prediction can be applied here to improve the performance by predicting the input data for the speculative thread. We developed a novel technique to achieve such

T. Srikanthan et al. (Eds.): ACSAC 2005, LNCS 3740, pp. 367 – 388, 2005.

value prediction efficiently and effectively in software. Our *software value prediction* (SVP) technique is implemented and evaluated in a speculative parallel threading (SPT) framework [2]. Before we describe the complete SVP methodology and its implementation in details, we first give a brief introduction of the SPT execution model and its application to loops.

1.1.1 The SPT Execution Model

In the SPT execution model that we studied [9], there are two processing units. One is designated to be the main processor that always runs in normal (non-speculative) mode. The other is speculative processor that runs in speculative mode. When the main thread on the main processor executes a special instruction *spt_fork*, a speculative thread is spawned with the context of the main thread on the speculative processor and starts running at the instruction address specified by *spt_fork*. All execution results of the speculative thread are buffered and not part of the program state. After executing the *spt_fork* instruction, the main thread continues its execution in parallel with the speculative thread. There is no direct communication or explicit synchronization between the two threads except they share the memory hierarchy. The instruction address of *spt_fork* in main thread is called *fork-point*; the address specified by *spt_fork* where the speculative thread begins its execution is *start-point*.

When the main thread arrives at the *start-point*, i.e., the place where the speculative thread starts the speculative execution, it will check the execution results of the speculative thread for any dependence violation. Depending on the check result, the main thread takes either of the following actions: If there is no dependence violation (i.e., no misspeculation), the entire speculative state is committed at once. Otherwise, the main thread will re-execute the misspeculated instructions while committing the correct execution results.

1.1.2 SPT Loop and Partition

We apply the SPT execution model to exploit loop-level parallelism. That is, when an iteration of a loop runs on the main processor, a speculative thread will be spawned on the speculative processor to run the successive iteration. We call this kind of loop

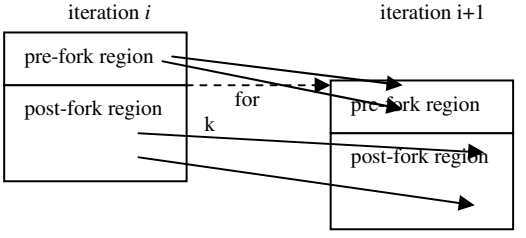


Fig. 1. SPT loop partition and execution scenario

(The arrows show the true dependences between the iterations. The middle line in the loop body refers to the fork-point, and the start-point is always at the beginning of the loop body.)

an *SPT loop*. The compiler's job is to generate SPT loops automatically and guarantee the SPT loops bring performance benefits. Fig. 1 illustrates the execution scenario of an SPT loop.

In Fig. 1, the main thread executing iteration *i* forks the speculative thread for iteration *i+1*. The insertion of the instruction *spt_fork* effectively partitions the loop body into two regions: a *pre-fork* region and a *post-fork* region. The true data dependences between two consecutive iterations are shown as arrows. We call the source of any cross-iteration dependence a *violation candidate*.

Since the speculative thread is spawned after the main thread has finished its pre-fork region, all dependences originated from the pre-fork region to the speculative thread are guaranteed to be satisfied. We care only those dependences originated from the post-fork region of the main thread. In order to avoid dependence violation, the compiler can try to reorder all violation candidates into pre-fork region. However, such code reordering will be restricted by the dependences between the pre-fork and the post-fork regions within the iteration. Furthermore, the pre-fork region is part of the sequential component of the parallel execution of an SPT loop. Amdahl's law requires the pre-fork region size small enough compared to the post-fork region in order to bring any parallelism benefits. Since the *start-point* of an SPT loop is always at the beginning of the loop body, the compiler's work for the SPT loop parallelization is essentially to perform desirable code transformations on the loop body and then determine the optimal *fork-point* within the loop body.

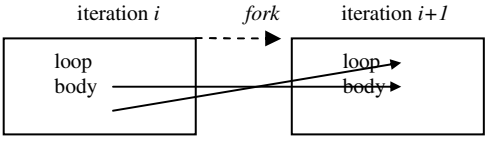
1.2 Software Value Prediction (SVP)

Value prediction techniques can be used to improve speculative thread-level parallelism. When a variable accessed by the speculative thread is probably to be modified by the main thread, we can predict the modified value for the speculative thread before its first access. If the value is correctly predicted, the corresponding data dependence is effectively removed, with certain overhead for the prediction though.

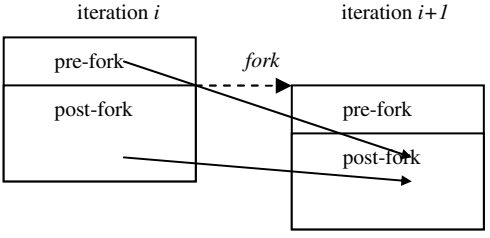
In our work, value prediction is conducted in pure software without any special value prediction hardware support. The idea of *software value prediction* (SVP) is to have the compiler determine which values are critical for performance and predictable, then inserting the prediction statements (predictors) in proper places in the compiled code. Fig. 2 illustrates how SVP is applied in SPT loop to improve the thread-level parallelisms. Fig. 2 (a) is a loop selected as SPT loop candidate; it has two violation candidates that defines values of variables `x` and `y` for next iteration. In Fig. 2 (b), the loop is transformed into an SPT loop where the violation candidate for variable `y` is moved into pre-fork region. Variable `x`'s definition cannot be reordered into pre-fork region because of either the intra-iteration dependence or the pre-fork region size constraints.

Fig. 2 (c) shows the defined value of variable `x` is predicted in pre-fork region. The original dependence from `x` is virtually replaced by the new dependence originated from variable `pred_x`.

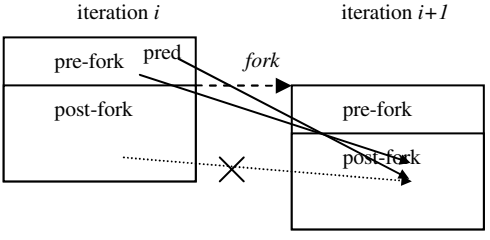
Different from the case of variable `y`, the value of `pred_x` may be mispredicted, causing misspeculation on variable `x`. The value misprediction will be captured by the software checking code inserted by the compiler, which we will explain next.



(a) An SPT loop candidate



(b) SPT loop without SVP applied



(c) SPT loop with SVP

Fig. 2. Illustration of SVP in SPT loop

1.2.1 An Example of Software Value Prediction

```
while(x){
   foo(x);
   x = bar(x);
}
```

(a) Original loop

```
pred_x = x
while(x){
 start:
   x = pred_x;
   pred_x = x+2;
   spt_fork(start);
   foo(x);
   x = bar(x);
   if(pred_x != x)
       pred_x = x;
}
```

(b) SVP-transformed

Fig. 3. Example of SVP compilation technique

Fig. 3 gives a more concrete example of the software value prediction technique.

Consider the loop in Fig 3 (a). It is very difficult to get any loop-level parallelism without any special transformation because there are cross iteration dependences inc-urred by variable `x`. Even with speculation, if the speculative thread uses old `x` value to compute `foo(x)`

and `bar(x)`, its speculative execution is useless. In this case, the compiler can profile the value `x`. Assuming it finds that `x` is often incremented by `2` by `bar(x)`, the compiler can predict `x` by inserting the predictor statement "`pred_x = x+2;`" before the *spt_fork* instruction as shown in Fig. 3 (b). The predicted value is used as the initial value of `x` by the speculation thread with statement "x = `pred_x;`". The check and recovery code "`if(pred_x != x) pred_x = x;`" is inserted at the end of loop.

If the predicted value `pred_x` does not match with the new value of `x`, the assignment statement "`pred_x = x;`" will be executed. As the variable `pred_x` becomes modified after the fork-point, it will be caught by the SPT dependence check as a dependence violation and triggers misspeculation recovery according to the SPT execution model. The re-execution of the next iteration is guaranteed to be correct because the value of `pred_x` has been corrected by "`pred_x = x;`".

Our study showed that software value prediction can boost thread-level speculation performance significantly. We have developed a complete and versatile SVP framework and implemented it in our SPT compiler. Our evaluation results confirm quantitatively that value prediction is essential to thread-level speculative parallelism. This paper examines the SVP technique in details, describes our general SVP framework and reports its effectiveness and efficiency in improving thread-level parallelism for the SPECint2000 benchmark suite.

1.3 Paper Organization

The rest of the paper is organized as following. Section 0 discusses the related work; then we briefly introduce the SPT compilation framework for loop iteration-based speculative thread-level parallelism in Section 0. Our general SVP framework and its detailed implementation in the SPT compiler are described in Section 0 and Section 0 respectively. Section 0 evaluates the SVP technique with its performance results and overhead data. We conclude the paper in Section 0.

2 Related Work

Lipasti, Wilkerson and Shen showed that load-value prediction is a promising technique to exceed data-flow limit in exploiting instruction level parallelism 11]. Other work showed that the value prediction mechanism could be applied to not only load instructions, but also nearly all value-generated instructions 5].

Most of current value prediction mechanisms studied in literatures 15],19],20],21],8] use special hardware to decide prediction pattern and to predict values. Basically there are four kinds of hardware value predictors: Last-value 11], Constant-stride [5], Context-based 18], and Hybrid 21]. The difficulty in hardware value prediction is in finding out whether an instruction is appropriate for prediction at runtime within acceptable period while keeping the hardware cost reasonable. In order to alleviate the severity of the problem, some work 7], 19] require compiler and/or profiling support such that only values that have high confidence are predicted.

Fu et al. 4] proposed a software-only value prediction approach that does not require any prediction hardware, and can run on existing microprocessors. It utilizes branch instruction for speculation: When the verification code finds a wrong

prediction, the control flow jumps to the recovery code. The prediction codes are statically inserted into the executables and are executed regardless whether the prediction is correct or necessary. Without special speculation hardware support, the execution context of the original code could be polluted by the prediction code. This approach showed limited performance improvement.

Fu et al. 3] also proposed to add explicit value prediction instructions in the instruction set architecture (ISA) for their value speculation scheduling.

Compiler-controlled value prediction optimization done by Larson and Austin 12] achieved better performance than previous pure software approach. It employs branch predictor for confidence estimation and uses the branch prediction accuracy to resolve the value prediction accuracy problem.

Zhai et al. 22] proposed compiler optimization for scalar value communication in speculative parallel threaded computation. Their compiler inserted synchronizations in critical forwarding paths in order to avoid speculation failure. Different from their approach, we do not insert synchronization to communicate correct values between threads; instead, we try to pre-compute and predict the values for speculative thread, so there is no extra synchronization overhead while may have misprediction penalty.

Steffen et al.[23] have proposed techniques to improve value communications in TLS computation, including value prediction for both memory values and forwarded values. They found predictors must be throttled to target only those dependences that limit performance, and squashing the prevalent silent stores can greatly improve the TLS performance. In SVP, the compiler only inserts predictors for the variables that are sure to bring performance benefits with our cost model. The last-value predictors in SVP is very similar to the silent stores, and we drew the same conclusion that last-value predictor contributes a large portion to the performance improvement.

Cintra and Torrellas[24] has proposed a complete hardware framework for handling data dependence violations in speculative parallelization. The techniques include value prediction for same-word dependences. They conducted the experiments with a suite of floating-point applications, and found that values are either highly unpredictable or do not change because they are accessed by silent stores. Our work with SVP is evaluated with SPECint2000 benchmarks, which shows value prediction is effective to eliminate squashes due to data dependences.

3 A Compilation Framework for Thread-Level Speculative Parallelization

We have developed a comprehensive compiler framework based on Open Research Compiler (ORC 16]) to extract loop-based thread-level speculative parallelism by executing successive iterations in parallel threads 2]. We developed a misspeculation cost model and used it to drive the speculative parallelization. The framework allows the use of enabling techniques such as loop unrolling, variable privatization and dependence profiling to expose more speculative thread-level parallelisms. Our software value prediction technique is implemented and evaluated in the framework. In this section, we give a brief description about our SPT compilation framework.

3.1 Cost-Driven Compilation

The goal of SPT compiler is to find out the optimal loop partition for each loop candidate so that the generated SPT loop has minimal misspeculation cost. SPT compiler accomplishes the goal with its cost-driven compilation framework in a two-pass compilation process, as is described in this section.

In order to estimate the misspeculation cost, we introduce the concepts of *dependence probability* and *misspeculation penalty*. Dependence probability $Probability_{Dependence}(dep)$ of a loop-carried dependence *dep* is the probability of the dependence really happening at runtime. It can be computed as the ratio of the number of iterations when the dependence happens to the number of total iterations the loop executes. The misspeculation penalty $Penalty_{Dependence}(dep)$ of the dependence is the misspeculated instruction count caused by the dependence and need to be re-executed. The *misspeculation penalty* caused by the dependence is computed as equation E1:

$$Penalty_{Misspeculation}(dep) = Penalty_{Dependence}(dep) * Probability_{Dependence}(dep) \quad \text{(E1)}$$

And the overall effects of all the dependences in a loop is computed as *misspeculation cost,* which is expressed conceptually[1] as equation E2,

$$Cost_{Misspeculation} = \sum_{dep} Penalty_{Misspeculation}(dep) \quad \text{(E2)}$$

In order to select and transform only good SPT loops without missing any good ones, SPT compiler goes through two compilation passes. The first pass selects loop candidates according to simple selection criteria like loop body size and trip count, and apply loop preprocessing such as loop unrolling and privatization for more opportunities of thread-level parallelism. Next, the SPT compiler finds out the optimal loop partition for each loop candidate, and determines its potential speculative parallelism amount. The optimal partition results of all loop candidates are output and the first pass finishes without any real permanent transformation. Then the second pass reads back the partition results and evaluates all loops together, selects all good and only good SPT loops. These loops are again preprocessed, partitioned, and transformed so as to generate final SPT code.

3.2 Optimal Loop Partition Searching and SPT Loop Transformation

Since a partition is decided by the set of statements in post-fork region (or pre-fork region because they are complementary), the misspeculation cost of a given partition is decided uniquely by the violation candidates in the post-fork region. That means a combination of violation candidates can uniquely decide a loop partition. So SPT compiler searches for the optimal loop partition of an SPT loop candidate by enumerating all the combinations of violation candidates, computing each combination's misspeculation cost assuming only the violation candidates in the combination are in post-fork region, and selecting the combination that has minimal cost. The actual search space is greatly reduced with bounding functions and heuristics 2].

[1] It is conceptual because the different dependences may cause same set of instructions to be re-executed. Please refer to 2] for detail computation of misspeculation cost.

After the first compilation pass, we obtain the optimal partition and its associated optimal misspeculation costs for each loop candidate. In the second compilation pass, our speculative parallelization examines all loop candidates together (such as all nesting levels of a loop nest) and select all those good SPT loops that likely bring performance benefits and does the final SPT transformation to generate optimal SPT loop code.

```
  while( c!= NULL ){
       c1 = c->next;
       free_Tconnector(c->c);
       xfree(c, sizeof(Clause));
       c = c1;
  }
(a)    original loop
  while( c!= NULL ){
  start:
       c = temp_c;
       c1 = c->next;
       temp_c = c1;
       SPT_FORK(start);
       free_Tconnector(c->c);
       xfree(c, sizeof(Clause));
       c = c1;
  }
  (b) SPT-transformed loop
```

Fig. 4. Example of SPT transformation (without SVP)

With the resulted optimal partition of a good SPT loop candidate, SPT compiler transforms the original loop to generate the expected partition by code reordering: Assuming the SPT_FORK statement (which is compiled into the *spt_fork* instruction later) is initially inserted at the loop body start, the transformation work is to move the violation candidates in the pre-fork region above the SPT_FORK statement.

Fig. 4 is a real example of SPT loop transformation from *parser* application in SPEC2000int benchmarks. In the example, the *start-point* is indicated by the label `start`. The loop body is partitioned by the statement `SPT_FORK(start)`. The violation candidate the is moved to pre-fork region is statement "`c = c1;`". Since variable `c1` depends on statement "`c1 = c->next;`", both are put in to pre-fork region. In order to maintain the live-range of variable `c`, another variable `temp_c` is introduced.

4 SVP Methodology in SPT Compiler

Software value prediction can be achieved systematically and effectively with SVP methodology we developed in SPT compiler. In this section, we use a practical example to explain the methodology. Our SVP implementation in SPT compiler will be described in Section 5, "SVP Implementation in SPT Compiler".

Fig. 5 is a code snippet in application *mcf* from SPECint2000 benchmark, which is a hot loop in function `price_out_impl()`, covering 21% of total running time with *train* input set. It is easy to realize that, this loop can barely deliver any iteration-based thread-level parallelisms if without any transformations because of the cross-

```
1:while( arcin ){
2:  tail = arcin->tail;
3:  if( tail->time + arcin->org_cost > latest ){
4:     arcin = (arc_t *)tail->mark;
5:     continue;
6:  }
7:  red_cost = compute_red_cost( arc_cost, tail,
                    head_potential );
8:  if( red_cost < 0 ){
9:    if( new_arcs < MAX_NEW_ARCS ){
10:      insert_new_arc( arcnew, new_arcs, tail,
                     head, arc_cost, red_cost );
11:      new_arcs++;
12:   }
13:   else if((cost_t)arcnew[0].flow > red_cost )
14:     replace_weaker_arc( arcnew, tail,
                       head, arc_cost, red_cost );
15:    }
16:    arcin = (arc_t *)tail->mark;
17:}
```

Fig. 5. Example loop in mcf from SPECint2000

iteration dependence incurred by variable `arcin` and `newarcs`. We will describe how SVP can improve the achieved parallelisms.

SPT compiler firstly identifies all the violation candidates, and estimates the misspeculation cost of them. For those dependences that have significantly impact on the delivered TLP, the compiler will figure out if there are opportunities to apply SVP technique, by predicting the dependence source variables. These variables are called *critical variables*, such as variable `arcin` and `new_arcs` in Fig. 5.

4.1 SPT Cost Model with SVP

The idea of software value prediction can fit in our SPT cost model described in Section 0 *3.1 Cost-Driven* Compilation very well. For example in Fig. 2, the original dependence caused by variable `x` in the loop without SVP is replaced by a new dependence originated from the predicted variable `pred_x` in pre-fork region. Since the prediction code in the main thread is executed before forking, the dependence from prediction can always be satisfied. The predicted value can be wrong in some iterations, which is characterized by the *probability of misprediction*; and the *misprediction penalty* of a dependence *dep* is the product of the misprediction probability and its dependence penalty shown as equation E3.

$$Penalty_{Misprediction} = Penalty_{Dependence}(dep) * Probability_{Misprediction}(dep) \quad \text{(E3)}$$

As long as the misprediction probability is much smaller than the replaced dependence probability and the predictor code size is small, SPT compiler can apply the prediction in loop transformation[2].

With the SVP cost model, it's clear that SVP does not try to reduce the dependence penalty of any dependence; instead, it tries to reduce the dependence probability to misprediction probability. Next we show how the model directs the software value prediction.

[2] The actual decision as to whether replace a dependence with prediction is made by computing the overall effect of the dependence or prediction penalties of all violation candidates.

After the compiler identifies the critical variables of the loop, SPT compiler needs to know the proper prediction pattern of each variable and the corresponding misprediction probability.

4.2 Selective Value Profiling

Prediction pattern of the critical variables could be found by program analysis, but it does not always work. For instance, neither `arcin` nor `new_arcs`' prediction pattern can be derived by program analysis because of complicated control and data flow. SVP uses value profiling for prediction pattern identification.

Only critical variables are profiled. The compiler instruments the application with hooks into a value profiling library, which has some built-in prediction patterns, such as:

- *constant*, where the variable has only one fixed value;
- *last-value*, where the variable has the same value as last time it is accessed;
- *constant-stride*, where the variable's value has constant difference with the value last time it is accessed;
- *bit-shift*, where the variable's value can be got by shifting its last value in constant bits;
- *multiplication*, where the variable's value is constant times of its last value;

The instrumented executable runs with typical input set for value profiling. During the value-profiling run, the runtime values of the critical variables are fed into the library, where they are matched with the built-in prediction patterns, and the matching results are recorded. In the end of the execution, the results are output into a feedback file which has the SVP-needed information for each critical variable, such as the best matched prediction pattern, the matching ratio with the pattern, and total hit counts, etc. The misprediction probability of the pattern is equal to (1 – matching ratio).

As to our example, when run with *train* input set, variable `arcin` shows constant-stride pattern with constant "`-192`" and perfect matching ratio 100%, while `new_arcs` shows last-value pattern with matching ratio 99%.

SPT compiler estimates the overall effects of the penalties of dependence or prediction of all violation candidates and decides which variables are to be predicted according to the cost model. Here both `arcin` and `new_arcs` are selected to be predicted.

4.3 SVP Code Transformation

After the predictable variables are selected, the compiler inserts the predictor statement in the loop by code transformation.

SVP transformation can be accomplished basically in four steps, although the real work depends on the speculative architecture and application model. Assume the *fork-point* and *start-point* is decided already and we are going to insert a predictor for variable `x` which has prediction pattern `predictor(x)`. A new variable `pred_x` is introduced to store the predicted value for the variable `x` in the master thread. The transformation steps are:

1. **Prediction initialization**: Insert an assignment "`pred_x = x;`" before the loop, which prepares the predicted value;
2. **Prediction use**: Insert an assignment "`x = pred_x;`" in the beginning of the loop body, which consumes the predicted value;
3. **Prediction generation**: Inserts an assignment "`pred_x = predictor(x);`" after the prediction use statement, which generates the predicted value and saves it in `pred_x`;
4. **Prediction verification**: Inserts statement "`if(pred_x!=x) pred_x = x;`" in the end of the loop body, which checks the correctness of the prediction and corrects it if it is wrong.

The SVP-transformed code for the example loop is shown in Fig. 6 Careful readers may find that we do not predict variable `new_arcs` with last-value predictor in the transformed code. The reason is, its last value as part of the thread context is copied to the speculative thread naturally as the SPT architecture supports. If the architecture model does not support context copy, SVP needs to predict it as well. These issues are important for a practical implementation.

```
1: pred_arcin = arcin; //prediction intialization
2: while( arcin ){
3:  start:              //start-point
4:   arcin = pred_arcin;   //prediction use
5:   pred_arcin = arcin -192; //prediction generation
6:   SPT_FOOK(start);      //fork-point
7:   tail = arcin->tail;
8:   if( tail->time + arcin->org_cost > latest ){
9:     arcin = (arc_t *)tail->mark;
10:        continue;
11:  }
12:  red_cost = compute_red_cost( arc_cost, tail,
                          head_potential );
13:  if( red_cost < 0 ){
14:    if( new_arcs < MAX_NEW_ARCS ){
15:      insert_new_arc( arcnew, new_arcs, tail,
                 head, arc_cost, red_cost );
16:      new_arcs++;
17:    }
18:    else if( (cost_t)arcnew[0].flow > red_cost )
19:      replace_weaker_arc( arcnew, tail, head,
                       arc_cost, red_cost );
20:  }
21:  arcin = (arc_t *)tail->mark;
22:  if( pred_arcin!=arcin )//prediction verification
23:  pred_arcin = arcin; //misprediction recovery
24: }
```

Fig. 6. Example loop in mcf with SVP transformation

In our experiment with the example loop, the transformed one gets 49.76% performance speedup compared to the original one, which alone increases mcf's overall performance by 10% because of the big coverage of the loop.

4.4 Advanced SVP Techniques

In our experiments, the methodology described so far is enough in handling most of the cases in SPECint2000 benchmarks. In this subsection, we introduce more sophisticated techniques for software value prediction in order to exploit more parallelisms.

4.4.1 Indirect Predictor

One situation we meet is that not all critical variables exhibit obvious prediction patterns, for instance, a variable storing memory pointer to the nodes of a tree, whose runtime values are pretty randomly scattered in the heap space.

Fig. 7 shows a similar but different situation, which is a loop in function compress_block() of SPECint2000 application gzip. In this loop, variable dx is critical for iteration-based speculative parallelisms, but its value does not have good predictability.

```
1: do {
2:     if ((lx & 7) == 0) flag = flag_buf[fx++];
3:     lc = l_buf[lx++];
4:     if ((flag & 1) == 0) {
5:       send_code(lc, ltree);
6:       Tracecv(isgraph(lc), (stderr," '%c' ", lc));
7:     } else {
8:       code = length_code[lc];
9:       send_code(code+LITERALS+1, ltree);
10:      extra = extra_lbits[code];
11:      if (extra != 0) {
12:          lc -= base_length[code];
13:          send_bits(lc, extra);
14:      }
15:      dist = d_buf[dx++];
16:      code = d_code(dist);
17:      Assert (code < D_CODES, "bad d_code");
18:      send_code(code, dtree);
19:      extra = extra_dbits[code];
20:      if (extra != 0) {
20:          dist -= base_dist[code];
21:          send_bits(dist, extra);
22:      }
23:    }
24:    flag >>= 1;
25:} while (lx < last_lit);
```

Fig. 7. Indirect predictor example in gzip application

On the other hand, `dx`'s value depends on another critical variable `flag` by control-dependence shown in bold fonts in the figure. If `flag` has very good predictability, we can predict `dx`'s value based on `flag`'s prediction as long as the control-dependence has reasonable probability. This kind of predictor is identified by combining program analysis technique with software value prediction, and is called *indirect predictor* for distinction from the previous *direct predictor*.

In some cases, the critical variable is predictable by both direct and indirect predictor. The compiler needs to choose either of them to apply the code transformation. Predictor selection is then an interesting topic that we will describe in our SVP implementation in Section 0.

4.4.2 Speculative Precomputation

Another important technique in SVP compilation is *speculative precomputation*, which predicts a critical variable with code slice extracted from the program without depending on any other variable's prediction.

Let's take an example from SPECint2000 application *twolf's* function `term_newpos()` shown in Fig. 8 .

In this loop, variable `termptr` is highly critical for correct speculation, but its value exhibits very low predictability. And because of the probable alias between `termptr` and other modified pointer variables in the loop body, the compiler cannot guarantee "`termptr = termptr->nextterm;`" alone can generate correct value for next iteration.

On the other hand, the compiler finds the alias probability between `termptr` and other pointer variables are small by value profiling or other profiling tools if available. It can extract the code "`termptr = termptr->nextterm;`" from `termptr`'s all possible definition statements, use it as the predictor so as to speculatively precompute `termptr`'s value for the speculative thread.

```
1: for( termptr = antrmptr ; termptr ;
      termptr = termptr->nextterm ) {
2:      ttermptr = termptr->termptr ;
3:      ttermptr->flag = 1 ;
4:      ttermptr->newx=termptr->txpos[newaor/2] + xcenter ;
5:      dimptr = netarray[ termptr->net ] ;
6:      if( dimptr->dflag == 0 ) {
7:        dimptr->dflag = 1 ;
8:        dimptr->new_total = dimptr->old_total +
                    ttermptr->newx - ttermptr->xpos ;
9:      } else {
10:       dimptr->new_total+=ttermptr->newx-ttermptr->xpos;
11:   }
12:}
```

Fig. 8. Speculative precomputation example in twolf

We find in our experiments that sometimes the predictor is the combination of speculative precomputation and the direct prediction. We regard this also a kind of indirect predictor.

4.4.3 Prediction Plan

Sometimes there are multiple critical variables that need predicting for one speculative thread, the compiler cannot simply group their predictors to be the prediction code, because these critical variables may be inter-dependent or their predictors may share some codes. A simple combination of the best predictors for each critical variable does not necessarily mean optimal prediction code, it is important to find out the combination that can achieve best overall TLP performance.

The prediction code for a speculative thread, as a reasonable combination of all the predictors, is called a *prediction plan*. It is desirable to find out the optimal prediction plan under our cost model that can bring minimal misspeculation cost with reasonable code size. We have developed such a plan-searching algorithm in our SVP implementation discussed in next section.

5 SVP Implementation in SPT Compiler

We introduced the SVP methodology in SPT compiler; now in this section we describe the implementation details of software value prediction about its critical variable identification and optimal prediction plan searching.

5.1 Critical Variable for Value Profiling

Since the predicted value is only used by the speculative thread in our model, it is irrelevant to the semantic correctness of the application. It could be possible to profile as more variables as possible and use more complicated prediction patterns in order to

achieve better performance; but we choose profile only some critical variables because of the two negative impacts caused by more profiled variables.

Although the speculation model guarantees the mispredicted values not affect the correctness, they do affect the performance. Prediction code run in the pre-fork region of the master thread is executed serially hence consumes precious processor cycles; too big size of the prediction code may negate our goal of increasing thread-level parallelisms. There is a tradeoff between the prediction code size and its reduced misspeculation penalty.

Actually SPT compiler has two thresholds for loop partitioning to count into the balance of pre-fork region size and misspeculation cost, that is, the pre-fork region size can never exceed size *S* and the misspeculation cost should be smaller than threshold *C*. Both thresholds are constants during the compilation as a ratio value to the loop body code size and execution time.

The code reordering in SPT compilation cannot increase the pre-fork region size to exceed *S* by moving a violation candidate into the pre-fork region, even the resulted loop partition has higher than *C* misspeculation cost. That is why the compiler fails to move the violation candidate x into pre-fork region in Fig. 2, even if it has high misspeculation penalty; and that is why value prediction is important: It can reduce the misspeculation cost of a violation candidate without moving it into pre-fork region like it does with variable x in Fig. 2. Actually in our evaluation with SPECint2000, we believe value prediction is essential for ultimate TLP performance.

SPT compiler introduces *dependence slice* in order to accurately estimate the code size for a violation candidate, which refers to the statements or expressions dependent by the violation candidate between *fork-point* and *start-point*, i.e., the loop body in our implementation.

As discussed above, when the dependence slice is very small, it could be cloned in the pre-fork region to precompute the value, which can be implemented by the speculative precomputation technique developed in SVP, while the precomputed value is always correct. On the other hand, since SPT compiler itself can reorder the small dependence slice to pre-fork region, there is no need to precompute it at the beginning. In any case, SVP will not profile this kind of defined variables.

If a variable has big dependence slice, it still does not necessarily mean to be profiled by SVP. One reason is the misspeculation penalty caused by the violation candidate can be very small compared to the cost threshold *C*. The other situation it is not profiled is, when the variable directly depends on another defined variable, we need profile only the latter one, and derive the former variable's value from their relation. This results with indirect predictor as we described in Subsection 0. In this case, the misprediction probability of the indirect predictor is derived by probability propagation in the dependence slice from the direct predictor.

Then all remaining critical variables for value profiling have high misspeculation cost and big dependence slice size, and we hope to predict them directly.

5.2 Prediction Plan Search

With the prediction pattern chosen for each critical variable by the profiling library, the compiler will decide how to predict it by prediction plan searching. We use a

branch-bound algorithm in our implementation to search for the optimal prediction plan. The pseudo-code of the algorithm is shown in function `predictor_plan_search()` in Fig. 9.

```
1: global var optimal_cost, optimal_plan;
2: predictor_plan_search( viol_cand_stack )
3: {
4:   path_end = TRUE;
5:   while( viol_cand_stack not empty ){
6:       viol_cand = viol_cand_stack.pop();

7:       for(each pred of viol_cand's predictors){
8:         pred_plan.add( pred );

9:         if( !exceed_size( pred_plan ) ){
10:              path_end = FALSE;
11:              predictor_plan_search(viol_cand_stack);
12:           }

13:           pred_plan.remove( pred );
14:         }//for
15:         if( exceed_cost( pred_plan ) )
16:           return;
17:      }//while

18:      if( path_end ){ //end of search path
19:        cost = misspec_cost(pred_plan);
20:      if( cost <= optimal_cost){
21:         optimal_cost = cost;
22:         optimal_plan = pred_plan;
         }
          }
```

Fig. 9. SVP prediction plan search algorithm

All the violation candidates are organized in a stack `viol_cand_stack`, which is the input argument of the recursive function. The function searches all the possible prediction plans recursively by examining the valid predictors of the violation candidates. Each search path enumerates the elements in the stack, computes the misspeculation cost of the valid predictor combinations of the popped violation candidates, and results with a prediction plan that records the prediction decision with each critical variable.

We use two bounding functions to effectively reduce the size of the searching space:

- `exceed_size ()`, which checks if the prediction code size exceeds the threshold *S* when a predictor is added into the prediction plan. If it is true, the predictor will not be used, and the search algorithm continues to check next available predictor or simply gives up predicting the variable;
- `exceed_cost()`, which checks if the misspeculation cost exceeds the threshold *C* when a violation candidate is decided not to be predicted. If it is true, the search path is stopped and the algorithm backtracks to other paths.

At first, all the violation candidates are stored in `viol_cand_stack` in the order according to their dependence relations, that is, if a violation candidate depends on

some other ones, it will not be pushed in the stack before the other ones are pushed already. This ordering is achievable because there is no cyclic dependence between the violation candidates (The cyclic dependent violation candidates will be treated as one candidate by the compiler in the first place). This stack layout prunes lots of unnecessary searching paths early because a violation candidate can not be indirectly predicted if its dependent candidate is not predicted, and the decision as to dependent candidates is guaranteed to be made already according to the stack layout.

The intermediate search result is kept in `pred_plan`, and the temporary optimal search result and its cost are kept in global variable `optimal_plan` and `optimal_cost`. We use a boolean variable `path_end` to indicate the end of a search path; and when meeting an end, i.e., a prediction plan is identified, the algorithm computes the misspeculation cost of the newly found plan, compares it with the temporary optimal result and chooses the smaller one to be the new optimal plan.

In the statement `7` of the pseudo-code, the compiler iterates through all the valid predictors of a violation candidate, including both the direct and the indirect predictors. For a critical variable `x`, its valid predictors are got basically by replacing the critical variables in its dependence slice one by one with their respective predictors. If the dependence slice is replaced completely with a prediction pattern, it is a direct-predictor; otherwise, it is an indirect-predictor. The real implementation is not so simple, but the idea keeps same.

6 Experiments and Results

We did experiments to evaluate the developed SVP technology with SPECint2000 benchmarks. The results with SPECint2000 benchmarks are encouraging, and demonstrate that software value prediction we developed is an effective and efficient technology to deliver TLP performance even without special value prediction hardware support.

6.1 Evaluation Methodology

We evaluated SVP with SPECint2000 benchmarks on our SPT simulator, which is an in-house data-flow IPF simulator developed for our SPT architecture and compiler research. It maintains two separate clocks, separate register states and speculative execution states to keep track of the simulated parallel speculative execution. It also simulates the shared memory/cache system and performs the necessary register and memory dependence checking. The architecture details can be found in [9]. Since our focus here is the SVP methodology and implementations, we didn't intend to explore and evaluate different architecture models for TLS in this paper. We used the current TLS model to demonstrate the SVP general methodology and effectiveness.

Since SPT compiler has implemented a code reordering algorithm to transform SPT loops, we use that as base line to study the incremental effect of SVP. Many induction variables are handled by code reordering without requiring value prediction. Without SVP, our compiler will reorder code to minimize cross-iteration data dependences. Only in cases where code reordering cannot be applied to reduce dependences, SVP is applied.

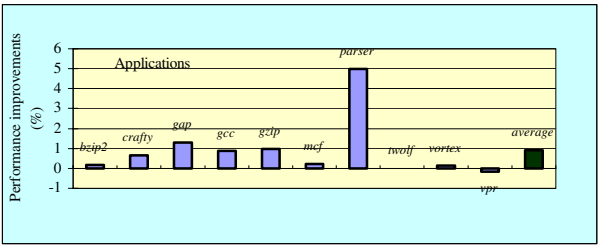


Fig. 10. Baseline SPT speedups without SVP

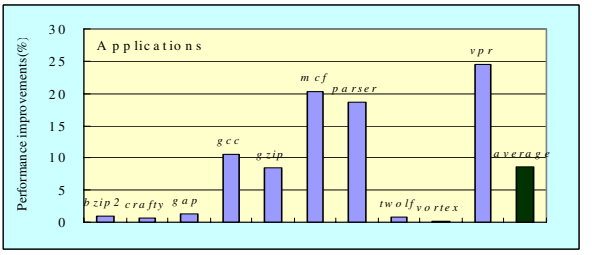
The performance improvements achieved by code reordering are given in Fig. 10. We can see that most applications get only marginal performance gains except *parser*, which has a close to 5.0% speedup; and the average improvement for all the ten applications is only 0.92%, almost negligible. *Vortex* is expected to have low speedup because it lacks appropriate loops for iteration-based speculative execution: The body size of its loops are too large for the processor store buffer to hold the temporary speculation results. It can be improved with region-based speculation, but is not this paper's focus.

6.2 TLP Performance with SVP

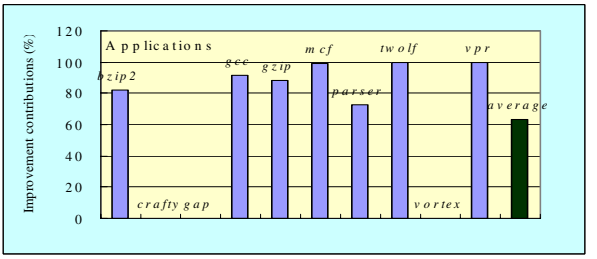
When SVP technology is applied, we get the speedups in Fig. 11. As clearly shown in the Fig. 11 (a), the average performance improvement is boosted from previous 0.91% to current 8.63%; and there are more than four applications have more than 10% speedups.

The contributions of SVP technology in total speedups achieved by SPT compiler is shown in Fig. 11 (b). For six of the ten SPECint2000 applications, SVP contributes more than 80% to the final performance; and the average contribution for all applications is 63%! This clearly demonstrates the essential effects of value prediction in delivering TLP performance.

Although SVP can improve the TLP performance dramatically, there are still four applications except *vortex* showing less than 4% speedups. The reason is our SVP technology is not fully tuned in SPT compiler, so some important loops are not transformed. In order to understand the potential of SVP technology, we applied it manually with some of the remaining loops in SPECint2000 applications. The manual transformations in our study were applied because the current ORC compiler did not support the needed optimization or analysis. We could implement those optimizations and analyses if time allowed. For example, in order to best transform a loop in "twolf", the compiler needs to resolve memory aliasing across procedure boundaries. The current ORC does not provide that support. However, this can be done either by inlining the corresponding function or by type-based inter-procedural alias analysis. We were very careful in deciding what manual transformation that we could apply. We only apply those that are practically feasible. Fig 12 shows the resulted potential speedups.



(a) Speedups with SVP technique



(b) SVP's contributions in total speedups

Fig. 11. Speedups with SVP and its contributions

In Fig. 12, all the applications except *vortex* achieve higher than 4% performance improvement; and the average potential speedup can be more than 15% even with *vortex* counted.

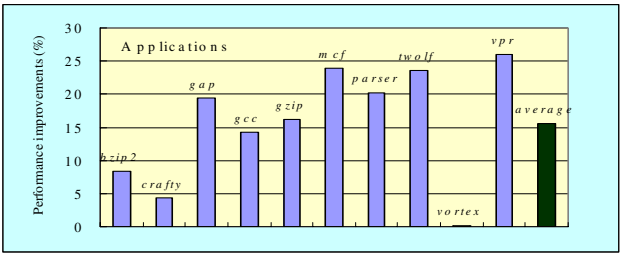


Fig. 12. Potential speedups with SVP technique

6.3 SVP Runtime Overhead

SVP conducts the value prediction purely in software, so it could be interesting to see how much runtime overhead it incurs. We examine the overhead by running the SVP-transformed executables sequentially, and then compare the running time with that of non-SVP-transformed executables.

The result show that the runtime overhead of SVP is marginal and almost negligible for most of the applications. The average overhead is less than 1%, meaning SVP is efficient in accomplishing its goal. We found it is interesting that *gap* and *twolf* have even smaller running time with the SVP-transformed executable,

which is contra-intuitive in the first glance but is actually reasonable, because we found the prediction code effectively prefetches some memory variables so that the cache misses have been reduced.

6.4 Contributions of Different Predictors

We want to know how different predictors are used in the applications, so we collect the contributions of different predictors to the total speedups. We classify the predictors into five categories.

- "const-stride" refers to the contribution is made by the loops that are transformed with only constant-stride predictors;
- "last-value + const-stride": with both last-value and constant-stride predictors, and without any other predictors;
- "indirect-predictor": with only indirect-predictors;
- "code-reordering": refers to the contribution is by SPT code reordering without prediction;
- "others": with other predictors that can not be classified into the above four categories.

The data in Fig. 13 shows that, last-value and constant-stride predictors are mostly important in all predictors: They in combination (const-stride and last-value+const-stride) contribute more than 60% to the total speedups. Applications *parser* and *twolf* achieve their performance mainly through indirect-predictor, which is because the predicted variables in these two applications are mostly pointer-chasing variables.

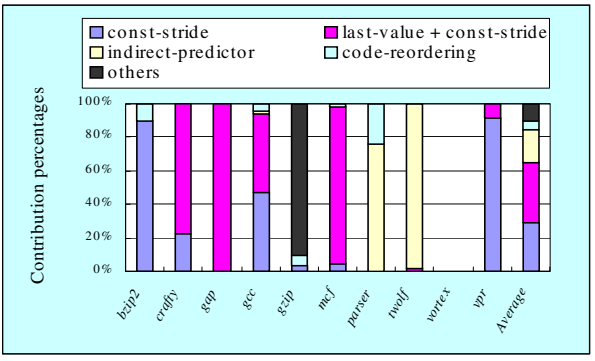


Fig. 13. Contributions of different predictors to total speedups

The data also shows the importance of direct-predictor over indirect-predictor, meaning the critical variables largely have simple prediction patterns they can be identified by value profiling directly.

Code-reordering alone contributes only a minor portion, which again demonstrates the essential value of value prediction in achieving good TLP performance.

6.5 Value Profiling Efficiency

In our experiments, the average number of the profiled variables per loop is only 1.72, which is pretty small. The profiling time increases significantly with an average of 105.79 times of the original application execution time. One important reason is, we profiled lots of variables that are considered by the compiler as critical variables just

because they are aliased with some critical variables, which can be improved by enhancing the alias analysis ability of the compiler. The profiling efficiency can further be improved by sample profiling.

6.6 Input Set Sensitivity

We used the "train" input sets for profiling. The results reported so far in the paper also used the "train" input set for evaluation. This corresponds to the ideal value prediction achievable by our SVP methodology and answers the ultimate performance question (i.e., on the best performance gain with SVP). We are fully aware of the input-set sensitivity problem which comes immediately after the performance question. So far we did have reasonable evidence that our results and conclusions are not sensitive to the input set used, though not strong enough.

We have run the same generated code but using the "ref" input sets and collected the performance results for seven Spec2000Int benchmark programs (Because of the time and resource issues, we are still collecting the rest ones.) Below in Table 1 is the comparison of the performance gains between those using "train" input sets and those using the "ref" input set. As expected, the performance gains using "ref" input sets are not as good as in the ideal case mostly. However, the differences are not large actually: the average speedup difference is only 7.79%, meaning the value predictability of the selected variables remains similar with different input sets.

Table 1. SVP speedup sensitivity to different input sets

	crafty	***gap***	***mcf***	***parser***	***twolf***	***vortex***	***vpr***	**average**
with train (%)	4.84	19.4	23.95	20.18	23.53	0.14	25.96	16.86
with ref (%)	3.69	19.4	19.71	16.96	25.81	0.14	23.1	15.54
diff (%)	23.76	0.00	17.70	15.96	-9.69	0.00	11.02	7.79

Basically our initial finding is, the different input sets do bring some differences in prediction accuracy, but not significantly. We studied the source code for the reason, and found most of the important predicted variables are either scalars or pointers.

The scalars are normally changed in a certain pattern (increment or bit-shift, for examples) under some conditions. If the condition is evaluated to be true in many iterations, the scalar can be predicted with the pattern; otherwise, it can be predicted with last-value pattern. Well under the two different input sets of our study, the condition evaluation result does not show big different trends. That means the same prediction pattern is applicable for both.

The pointer values are mainly decided by the memory management library which is the same for both input sets, so the prediction patterns of them are not changed much either.

We are not arguing the SVP technique is insensitive to different input sets, and we are looking for alternative approaches that can inherently eliminate the sensitivity issue.

7 Conclusions

Value prediction has been considered to be a valid approach to break data-flow parallelism limit. In this paper, we demonstrate that, value prediction is actually essential for the speculative thread-level parallelism, and can be achieved purely in software without any prediction hardware support. We show that, software value prediction can be achieved systematically and effectively by SVP compilation technology, which brings up to 15.63% performance improvement on average for SPECint2000 benchmarks with loop-based thread-level speculation. We also studied the contributions of different predictors, and found last-value and constant-stride predictors are most important for good speedups.

SVP compilation technology can be applied in areas beyond the loop speculation, such as region speculation, call-continuation speculation, etc. And it is not necessarily beneficial only to thread-level parallelism. For example as our data shown with *gap* application, SVP can effectively reduce cache misses so as to improve even the sequential execution performance. In SPT compiler, value profiling and SVP transformation are both carried in static compilation, we are also researching in dynamic profiling and just-in-time SVP transformation.

References

1. B. Calder, G. Reinman, and D. Tullsen, "Selective Value Prediction", International Symposium on Computer Architecture, 1999.
2. Zhao-Hui Du, Chu-Cheow Lim, Xiao-Feng Li, Chen Yang, Qingyu Zhao and Tin-Fook Ngai, “A Cost-Driven Compilation Framework for Speculative Parallelization of Sequential Programs”, in Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation, Washington, D.C., June 9-11, 2004.
3. C. Fu, M. D. Jennings, S. Y. Larin, and T. M. Conte, "Value Speculation Scheduling for High Performance Processors", in 8th International Conference on Architectural Support for Programming Languages and Operating Systems, October 1998.
4. C. Fu, M. D. Jennings, S. Y. Larin, and T. M. Conte, "Software-Only Value Speculation Scheduling", Technical Report, Dept. of Electrical and Computer Engineering, North Carolina State University, Raleigh, NC27695-7911, June 1998
5. F. Gabbay, "Speculative execution based on value prediction", Technical Report 1080, Department of Electrical Engineering, Technion-Israel Institute of Technology, 1996.
6. J. Gonzalez and A. Gonzalez, "The Potential of Data Value Speculation to Boost ILP"; International Conference on Supercomputing, 1998.
7. F. Gabbay and A. Mendelson, "Can Program Profiling Support Value Prediction", International Symposium on Microarchitecture, 1997.
8. Shiwen Hu, Ravi Bhargava, and Lizy K. John, “The role of Return Value Prediction in Exploiting Speculative Method-Level Parallelism”, The First Value-Prediction Workshop, San Diego, CA, June 7, 2003.
9. Xiao-Feng Li, Zhao-Hui Du, Chen Yang, Chu-Cheow Lim, Tin-Fook Ngai, “Speculative Parallel Threading Architecture and Compilation”, 4th Workshop on Compile and Runtime Techniques for Parallel Computing, Oslo, Norway, June 2005.

10. Xiao-Feng Li, Zhao-Hui Du, Qingyu Zhao, and Tin-Fook Ngai, “Software value prediction for speculative parallel threaded computations”, The First Value-Prediction Workshop, San Diego, CA, June 7, 2003.
11. M. Lipasti, C. Wilkerson and J. Shen, "Value Locality and Load Value Prediction", International Conference on Architectural Support for Programming Languages and Operating Systems, 1996.
12. E. Larson and T. Austin, "Compiler Controlled Value Prediction Using Branch Predictor Based Confidence", ACM/IEEE 33rd International Symposium on Microarchitecture (MICRO-33), December 2000.
13. P. Marcuello and A. Gonzalez, "A Quantitative Assessment of Thread-Level Speculation Techniques", Proc. of the 1st. Int. Parallel and Distributed Processing Symposium (IPDPS'00), Mexico, May 1-4, 2000.
14. P. Marcuello, J. Tubella, and A. Gonzalez; "Value Prediction for Speculative Multithreaded Processors"; International Symposium on Microarchitecture, 1999.
15. T. Nakra, R. Gupta, and M.L. Soffa; "Global Context-Based Value Prediction"; International Symposium on High-Performance Computer Architecture, 1999.
16. Open Research Compiler, Intel Co. Ltd., http://ipf-orc.sourceforge.net/.
17. J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry. A scalable approach to thread-level speculation. Proceedings of Intl Symp. On Computer Architecture 2000, pp. 1-24.
18. Y. Sazeides and J. Smith; "The Predicatibility of Data Values"; International Symposium on Microarchitecture, 1997.
19. Y. Sazeides and J. Smith, "Modeling Program Predictability", International Symposium on Computer Architecture, 1998.
20. R. Thomas and M. Franklin, "Using Dataflow Based Context for Accurate Value Prediction", Proc. International Conference on Parallel Architectures and Compilation Techniques (PACT), 2001.
21. K. Wang and M. Franklin, "Highly Accurate Data Value Prediction using Hybrid Predictors"; International Symposium on Microarchitecture, 1997.
22. A. Zhai, C. B. Colohan, J. G. Steffan and T. C. Mowry, “Compiler Optimization of Scalar Value Communication Between Speculative Threads”, The Tenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-X), San Jose, CA, USA, Oct 7-9, 2002.
23. J. G. Steffan, C. B. Colohan, A. Zhai, T, C. Mowry. "Improving Value Communication for Thread-Level Speculation" , The Eighth International Symposium on High-Performance Computer Architecture (HPCA'02), Boston, MA, USA, Feb 2002..
24. M. Cintra, J. Torrellas, "Eliminating Squashes Through Learning Cross-Thread Violations in Speculative Parallelization for Multiprocessors" The Eighth International Symposium on High-Performance Computer Architecture (HPCA'02), Boston, MA, USA, Feb 2002.

Challenges and Opportunities on Multi-core Microprocessor

Jesse Fang

Director and Chief Scientist of Programming System Lab,
Microprocessor Technology Labs,
Intel Corporation Technology Group
jesse.z.fang@intel.com

As Moore's Law predicates, the number of transistors on die will continue increase on die in future. It will be a big challenge for computer industry to effectively and efficiently use these transistors and optimize the microprocessor design under power envelop. Performance is not the only concern. The other design goals such as user friendly interface, easy for maintenance, security and reliability, will become more and more important. There is a clear trend in computer industry that more and more chip will implement multi-core on die.

In such multi-core design, the core itself may become simple for power and thermal consideration. The challenge to make such CPU chip success will be not rely on core design. It will depend on the uArch features to make these cores working efficiently such as cache hierarchy and interconnect topology. Such multi-core CPU chip is not simply implementing SMP (Shared Memory Processors) system on die. For instance, the on-die interconnect bandwidth between cores will be $>$ 100 Tera bytes/second. Considering that the shared memory bandwidth in today's SMP system is only $\sim$ 40 GB/s, the ratio between bandwidth and CPU frequency increases $>$ 200 times. Another example is communication latency between cores. In today SMP system, it is $\sim$ 400 cycles through the shared memory. In multi-core, the communication latency on die is about 20 cycles. It will improve near 20 times!

The challenges in the research for multi-core focus on scalability, programmability and reliability for future computer systems. A number of promising research results showed that the opportunities will be in integrating of software and hardware. Transactional Memory is a great example. It is able to provide an programming environment to guarantee atomicity, consistency and isolation for thread-level parallel applications, which can make programmer life much easier to write down parallel applications on such multi-core systems. On-die cache hierarchy and interconnect is another example for multi-core. Different algorithms in the applications may require different interconnect topology to optimize the performance. Simple shared-memory model or message passing model is hard to be used for all these apps. The interconnect design, or called "Microgrid on-die", will become important for both HW and SW. The Microgrid may need to be reconfigurable or programmable to meet the various needs of programming models for different parallel alls.

T. Srikanthan et al. (Eds.): ACSAC 2005, LNCS 3740, pp. 389–390, 2005.

Software is a huge challenge for the success of such multi-core chip. Research efforts have been devoted with the thread-level parallel processing for couple decades. The research results showed that the benefit of automatic parallel compiler was very limited even for high-performance computation. The parallel languages or language extensions like OpenMP and MPI only benefit certain groups of application programmers, but still have long way to become common for ordinary users. It is very challenging research to make programmers write down their parallel applications without thinking too much on the parallelism. But we believe it is the time right now that the computer industry and computer science research community need to focus on it again, because SW is much behind HW in the area.

Multi-core is the trend of microprocessor industry. More and more dual-core, four-core and even more cores on die will show in near future. It is challenging for computer industry to use these multi-core chip, but it is great opportunity also for computer research community.

Software-Oriented System-Level Simulation for Design Space Exploration of Reconfigurable Architectures

K.S. Tham and D.L. Maskell

Centre for High Performance Embedded Systems,
Nanyang Technological University, Singapore
{tham0014, asdouglas}@ntu.edu.sg

Abstract. To efficiently utilize the functionality of dynamic reconfigurable computing systems, it is imperative that a software-oriented approach for modeling complex hardware/software systems be adopted. To achieve this, we need to enhance the simulation environment to understand these dynamic reconfiguration requirements during system-level modeling. We present a flexible simulation model which is able to produce multiple views/contexts for a given problem. We use this model to examine each view for the mapping space, bandwidth, reconfiguration requirements, and configuration patterns of each computational problem.

1 Introduction

Dynamic reconfiguration of Field Programmable Gate Array (FPGA) devices has increased the flexibility of reconfigurable computing systems. However, to fully utilize dynamic reconfiguration requires more from the system host resource management [1] to deal with the complexity of scheduling hardware tasks in the fabric and to handle reconfiguration requests. As many different applications (e.g. digital signal processing (DSP) algorithms) consist of dedicated kernel tasks that are computationally intensive and have different mapping characteristics, this affects the resource allocation in the hardware and the host performance to meet the task's bandwidth requirements. This allocation can be temporal and will reuse the same physical space to reconfigure the hardware with different models to support the kernel tasks. As such, it will have an impact on the deployed scheduling method to conjure an appropriate schedule to organize the reconfiguration events. When modeling the hardware system, it would be useful if these hardware specific events can be considered as early as possible in the software generation phase during the system-level hardware/software co-design. Therefore, we need to enhance the design or simulation environment to encapsulate hardware-software constraints in an integrated design flow, so that we can profile the system behaviour and discover any hardware or software optimization.

With the current trend in embedded system design focusing on simplifying the software generation process while maintaining design quality, modeling of

T. Srikanthan et al. (Eds.): ACSAC 2005, LNCS 3740, pp. 391–404, 2005.

complex hardware/software systems are beginning to evolve from a software-oriented approach. SystemC [2] and Handel-C [3] are examples of modeling platforms consisting of software defined class libraries for system behavioural and register-transfer-level (RTL) designs. The objective of this paper is to demonstrate the benefit of customizing SystemC to enable modeling of different computational problems on various hardware structures with respect to their mapping space, bandwidth and reconfiguration requirements so that these attributes can be exploited by the scheduling algorithm. From these simulations, we can examine the configuration patterns suitable for device configuration.

SystemC provides a convenient platform as it is based on standard ANSI C++ and includes a simulation kernel for behavioural and RTL designs. It is a single-source approach, where the same source is used throughout the co-design flow. One of the major features, similar to VHDL, is that it supports design reuse through component libraries to create a structural hierarchical design. It is expected that SystemC will become more flexible and implement a complete software generation methodology on any real-time operating system (RTOS) for system-level modeling and embedded design [4]. This will allow the simulation and execution model to be identical and will ensure portability.

The next section of this paper describes the characteristics of reconfigurable architectures for our simulation approach. Then we introduce our simulation model for regular structure design. We present examples for mapping a set of DSP application tasks. In Section 4, we present resulting configuration patterns from our simulation that can be used for device reconfiguration. Finally we conclude our work and discuss future directions in Section 5.

2 Reconfigurable System Architectures

A reconfigurable architecture can adopt two types of reconfiguration characteristics; Compile time reconfiguration and run time reconfiguration. In the compile time method, the device configuration is loaded with a particular execution model that remains unchanged throughout the application life time and has a similar approach to ASIC. The MorphoSys [5], Raw [6] and Stream Architectures [7] are variants of this methodology, but their flexibility can be extended through programmable interconnects between hardware resources in the fabric. Our simulation approach will focus on the run time reconfiguration method, e.g. FPGA for custom computing machines (FCCM) where different hardware modules or cores can be implemented on the FPGA device [8]. As the device constantly changes its configuration, managing this type of system presents a greater challenge at run-time.

These hardware modules which contain kernel level operations will follow a particular execution model (e.g. systolic, SIMD, VLIW, etc) and exploit different levels of parallelism (e.g. instruction-level parallelism (ILP), data-level parallelism (DLP), etc). In order to support mapping of different tasks, most of these execution models consist of a group of regular hardware structures arranged in a specific pattern (to accelerate data or instructions) and have reconfiguration

features to change their resource interconnection or functionality. Therefore, instead of modeling multiple static cores, it is also beneficial for our simulation to have the flexibility of loading a number of such execution models (e.g. systolic 2D mesh array) that will match the computational and architectural granularity of the kernel task.

2.1 Reconfigurable by Design

Reconfigurable modules can consist of tasks/operations with different levels of granularity. We can describe the granularity as fine-grained, medium-grained and coarse-grained. Fine-grained tasks contain basic addition/subtraction and multiplication functions. Medium-grained tasks perform the function of the module, while coarse-grained are mainly system-level functions and are implemented by the operating system.

From a system-level, a hardware design can be partitioned into regions where reconfiguration can occur and regions that remain functionally static. The reconfigurable regions can be built from a hierarchy of medium-grained tasks or a compilation of fine-grained ones to form a medium-grained task. Building specialized circuits from a description at compile/run-time is a non-trivial process and has been an extensive area of research [9] [10]. This compilation process also requires support from the run-time system to place and route the circuits during run-time. This demanding process can be simplified by back-end tools [11] that support partial reconfiguration. However, the design must be subjected to some constraints in the placement and routing of the circuits, and are device-technology dependent. We take a simpler approach to generate a design by building from known hardware structures that have a well-defined map for the particular type of task to perform. Using these structures, we can use commercial back-end tools to analyse their physical mapping constraints and then feedback the constraints or results as input parameters for our system-level simulation.

These hardware structures can be reconfigured to support other tasks with the same degree of granularity. For example, the Discrete Cosine Transform (DCT) can be accomplished through matrix multiplication. There are several ways of implementing matrix multiplication in hardware but generally it is comprised of a collective of fine-grained operations (adders and multipliers) depending on the number of matrices to compute. The collective of fine-grained operations will then simultaneously perform the medium-grained operation. In this way, a software and hardware representation of the medium-grained operation can be easily produced through the definitions of pre-defined groups of lower level components.

Example: Matrix Multiplication. We can view matrix multiplication as a simple data-flow multiplication of matrices, constants and vectors. Using a systolic array structure of processing elements (PE), we can achieve two different hardware maps.

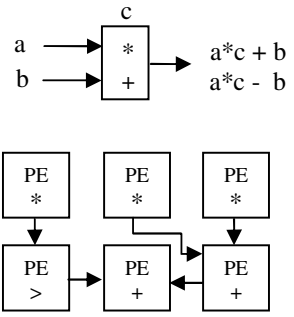


Fig. 1. PE functions and Matrix Multiplication mapping

Figure 1 shows the construction of a PE. Each PE is capable of multiplying by a constant and adding the result to an input operand. The hardware map for computing a single element in a 3x3 matrix multiplication using 6 PEs is also shown. We can see that this mapping is implicit to the matrix multiplication algorithm which simply involves 3 multiply, 2 add and 1 delay. Figure 2 shows an alternative mapping. By reordering the input, we can achieve a mapping that uses only 3 PEs, compared to six in the first mapping. In this mapping, the first result element of the 3x3 matrix multiplication will emerge at the end of the 3rd cycle. Thereafter each result element will be produced at every cycle. However, the inputs of the array need to be reordered and the data flow pattern is less obvious. An important feature of the second mapping is that the array can be extended to support larger multiplication by simply adding PEs to the right of the array. This is not the case for the first mapping example.

Example: DCT Using Matrix Multiplication. The DCT is used commonly in image processing applications especially in JPEG and MPEG compression techniques. We will consider one of the possible systolic architectures for implementing the 2-D DCT algorithm [12].

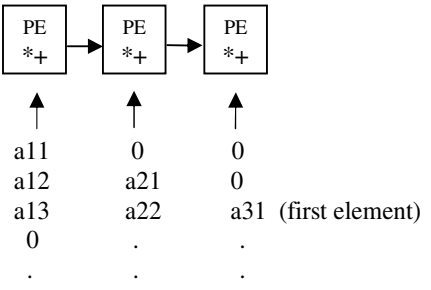


Fig. 2. Alternative Matrix Multiplication mapping

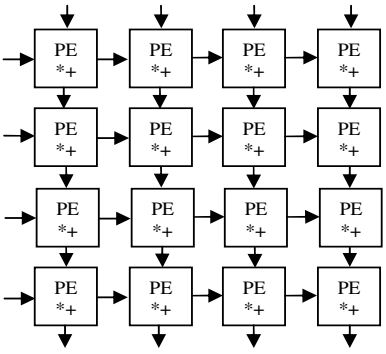


Fig. 3. 2D DCT mapping

Figure 3 shows the hardware map using a 4x4 matrix multiplication structure. It can be noted that this mapping has a similar construction to the mapping shown in Figure 2. The inputs to the array need to be interleaved and reordered so that a 2-D computation can be achieved.

From the above examples, we can see that medium grained operations can be composed of a group of fine-grained operations regular in function and interconnection, forming an execution model to exploit DLP. By looking at the mapping of the design, we can determine the configuration of each PEs, the computation cost and the corresponding mapping space. Our simulation, described in the next section presents multiple views of the resource to highlight these factors.

3 Characteristics of the Simulation Environment

In this section, we first describe a resource model that will be used for our case study. We then describe the specifics of the environment to incorporate multiple systolic 2D mesh arrays without recompiling them and how the software description will be translated into a hardware representation. Finally we demonstrate the flexibility of our simulation environment using matrix multiplication and DCT operations as examples.

3.1 Resource Model

Based on the examples highlighted in Section 2, our model consists of rows and columns of identical PEs. In Figure 1, the function of each PE is a multiply-add/sub operation. The multiply function can be disabled by multiplying by a constant 1. Likewise, the add function can be bypassed by adding a constant 0. This actually represents a delay unit. The multiplier architecture we will be using is based on the multiplier unit of the Systola 1024 parallel computer [13] which is a signed integer bit serial-parallel implementation. The significance of the bit-serial approach is the area advantage, but for an n-bit multiplier it requires 2n

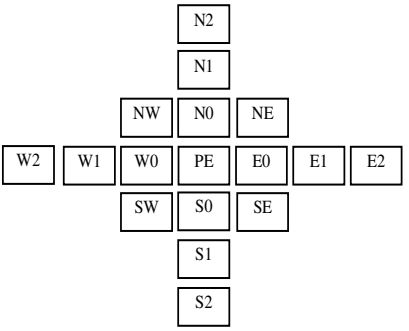


Fig. 4. PE interconnection pattern

cycles to complete. However, due to the concurrency of massively parallel PEs, the work load can be distributed across the fabric, with an additional speed-up from a higher clock frequency.

The interconnection between the PEs is indicated in Figure 4. Each PE has the capability of connecting to 16 of its nearest neighbours and each connection is a single-bit channel. This reconfigurable feature allows support for computational problems that have non-regular data flow patterns.

Figure 5 shows the block diagram of the PE design. The PE is similar to the functional unit of the MorphoSys architecture [5] except that it is of finer-granularity. The context register contains the multiplier constant k, and the multiplexer configuration for selecting any of its 16 neighbours. It can be noted that it would require a large bus to broadcast each PE's configuration and would be impractical. In the MorphoSys architecture, all PEs in a row or

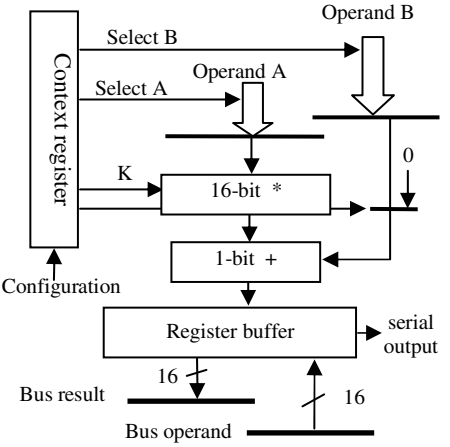


Fig. 5. PE block diagram

column have the same configuration data. Likewise, we use a bit channel to stream configuration data serially to each PEs. However, we can modify the array design to speed up this configuration. This technique will be described later.

The result can be stored in the buffer and output serially to neighbouring PEs or read directly from the result bus by the host. Similarly, the operand can be broadcast and loaded into the buffer through the operand bus. The computation result of the PE is represented by a sign extended 16-bit 2's complement number. This is to ensure that the range will be within 32-bits. Since the operands can only be 16-bit wide (stream serially), no overflow can occur.

SystemC PE and 2D Array RTL Design. We have created an RTL model in SystemC for the above PE and a 16 x 16 array structure (144 PEs). Figure 6 shows the hierarchy structure of the array. This hierarchy also shows how the components in the library can be reused. The library will contain basic components like full adder, register buffers and interface specifications. These components are further specified in terms of their construction using common components such as gates (AND, OR, XOR etc), flip-flops and latches etc. This makes the verification process easier as it can be performed at different levels of the hierarchy and isolated from the other levels.

The reuse capability can be achieved by the attributes of module inheritance and polymorphism associated with C++ to create regular structures. Once the regular structures are defined we can duplicate the interconnection of a single PE to construct the entire array. An excerpt of the SystemC code to implement the PE interconnection pattern depicted in Figure 4 is shown below.

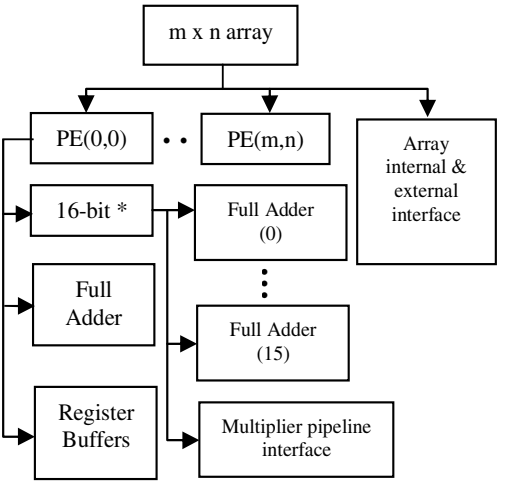


Fig. 6. Structural hierarchy of the array

```
//connect all PE
for (row) {
 for (col) {
    pe[cnt]->node_n0(peConn[row-1][col]);
    pe[cnt]->node_n1(peConn[row-2][col]);
    pe[cnt]->node_n2(peConn[row-3][col]);
    pe[cnt]->node_nw(peConn[row-1][col-1]);
    pe[cnt]->node_ne(peConn[row-1][col+1]);
    pe[cnt]->node_w0(peConn[row][col-1]);
    pe[cnt]->node_w1(peConn[row][col-2]);
    pe[cnt]->node_w2(peConn[row][col-3]);
    pe[cnt]->node_e0(peConn[row][col+1]);
    pe[cnt]->node_e1(peConn[row][col+2]);
    pe[cnt]->node_e2(peConn[row][col+3]);
    pe[cnt]->node_s0(peConn[row+1][col]);
    pe[cnt]->node_s1(peConn[row+2][col]);
    pe[cnt]->node_s2(peConn[row+3][col]);
    pe[cnt]->node_sw(peConn[row+1][col-1]);
    pe[cnt]->node_se(peConn[row+1][col+1]);
    pe[cnt]->result(peConn[row][col]);
 }
}
```

From the above, the array structure can be easily modified to any other type (e.g. linear array, tree, hypercube etc) by customizing the array interface description while the rest of the components can be reused. The array structure also needs to include an external interface to the host. For now, we will assume a simple memory mapped I/O model for communication to the array.

3.2 Simulation Model

We customize SystemC's simulation kernel to suit our simulation approach through a sequencer program. Our C++ sequencer program runs on top of the simulation kernel and contains interface information about the host and hardware cores (control and data). At compile-time, we can include different hardware cores in the simulation environment. In our case, the hardware cores represent different sizes of the 2D mesh array structures. At run-time, the sequencer can search for a known hardware representation for the particular hardware core and perform the mapping. It is able to multiplex between different cores. This multiplexing also applies to the host interface signals which will be shared among the hardware cores. However, if no sharing is allowed, additional host signals can be added into the sequencer program. The behaviour of the host is implemented by the sequencer program which controls the hardware cores directly. Currently, we do not target a particular host, like a RISC CPU or equivalent, however, the environment has the capability to do so.

The sequencer program does not automatically generate the hardware representation from the software routines. Our assumption is that some form of

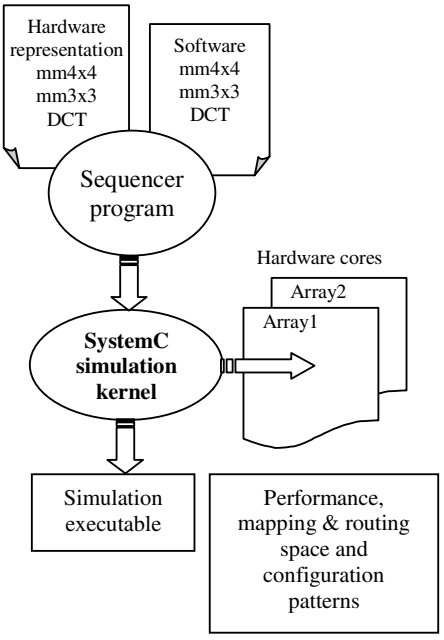


Fig. 7. Block diagram of the simulation environment

partitioning process has been applied and the representations are available to the simulation environment. However, we describe in the next section how the sequencer program can be customized to perform such a translation. This is purely a behavioural method but it actually resembles that of a decoder unit and the instruction set architecture (ISA) of a general purpose processor.

The construction of our simulation environment will produce multiple views/ contexts of the mapping of the computational problem with respect to the selected hardware array. It must be pointed out that the multiple views are in simulation space and not the physical hardware space. Although the mapping of the problem has a direct relationship to the physical space, at present our simulation only takes into consideration the execution model and is not device technology dependent. We have created a graphical display to show the mapping and routing space of the various views.

When more parallelism is desired, it is not always feasible to contain multiple arrays in the same physical space due to the massive number of PEs associated with systolic arrays. Thus, the physical hardware space will change context from time to time. This can be easily supported by our simulation model by switching context between the hardware cores. However, if different computational problems require different hardware cores to fit into the same physical space, it is still possible to simulate them concurrently. The sequencer program can control them separately as each hardware core will have a set of control and data signals e.g. execute, stop, config_PE(x, y) etc. Scaling the array to fit the problem can be

done by allocating unused portions of the array. However, a point to note is that allocating unused portions only applies to the context of the simulation space. To implement them in hardware would require various design restrictions which the run-time system must be able to handle. The resulting hardware-software representation can also be used to enhance the partitioning process.

Extending our simulation model is relatively simple. New hardware cores can be included and their interface can be incorporated into the sequencer program.

Hardware-Software Representation. As we have highlighted above, hardware -software partitioning is assumed to have already been applied. The example below shows the C source code for 4x4 matrix multiplication which has been identified for hardware implementation. Most compilers can achieve a speed up using the loop unrolling method which exploits ILP.

```
Example of a C code for 4x4 matrix multiplication

void mm (int mm_1[row*col], int mm_2[row*col]) {
    ...
    for(i=0; i<4; i++)
        for(j=0; j<4; j++)
            for(k=0, mm[i][j]=0; k<4; k++)
                mm[i][j] += mm_1[i][k] * mm_2[k][j];
}
```

The mm function declaration in C can be decoded into several customized instructions based on the matrix multiplication mapping detailed in Section 2.1. Basically the customized instructions need to control each PE for configuration, reading/writing the results/operands and executing start/stop command. The example below shows an excerpt of the pseudo hardware representation code.

```
Example of the hardware representation code

mmstruct hw_mm (int mm_1[row*col], int mm_2[row*col]) {
 ...
 //configuration array
 bus_config[0] = "001";
 ...

 //configure PEs
 for(int i=0; i<=row; i++) {
    for(int j=0; j<=col; col++){
        activate PE config signal;
        broadcast configuration;
        deactivate PE config signal
    ...
    }
 }
```

```
 //load data into PE
 for (i)
 {
    *(PE_DATA+i) = mm_1[i];
 }
 //execute
 send start_mm

 while (x cycle);
 //read data
 for(i)
    return *(PE_DATA+i);
}
```

3.3 Example: Application Mapping

To illustrate the application mapping process, we will use the examples described in Section 2.1. We will make the following assumptions:

1. A 16x16 2D mesh array is used
2. All PEs in the array are configured by a 1-bit channel in a serial fashion. It will take 26 cycles to configure a PE with a 26-bit wide context register
3. The host has a 4x16-bit data bus and each 16-bit data bus is assigned to every 4 columns of PEs
4. The PEs have a similar computation and I/O clock frequency

Table 1 shows the comparison of the number of PEs and cycles used for computing, performing I/O and configuration between the different operations. The comparison made here is to observe how the mapping space in regular structure design can be affected by the different implementations of the computation

Table 1. Number of PEs and cycles for computation, I/O and configuration

Operations	No.PEs	No.Cycles
4x4 Matrix multiplication mapping 1	Total: 144 Compute: 128 IO: 16	Config: 3744 Compute: 48 IO: 4
4x4 Matrix multiplication mapping 2	Total: 20 Compute: 16 IO: 4	Config: 520 Compute: 112 IO: 44
2D DCT (4x4 block)	Total: 20 Compute: 16 IO: 4	Config: 520 Compute: 176 IO: 76

problem as well as the host interface parameters. It is not used for benchmarking other micro-architectures. From these result, we can begin to suggest improvements in the array or host interface design. For example, from the first mapping of the matrix multiplication, the configuration cycle is more than 7 times that of the second mapping. This is because the first mapping uses 7 times more PEs. However, the compute cycle is more than 2 times faster, which is expected. To improve the first mapping we can either increase the configuration bandwidth or schedule configuration phases to inter-leave with other non-computational phases. It can be noted that the configuration cycles mentioned above are the time it takes to reconfigure the PE's function and connection to take up a new shape. The device configuration time and dynamic reconfiguration properties are treated separately for now. These will be described briefly in the next section.

Another point to observe is the I/O bandwidth. By allocating more PEs to service the I/O operations does not necessary reduce the time it takes to load or read the PEs. The bottleneck is still on the host data bus bandwidth and the layout of these buses across the array.

4 Configuration Patterns

From our simulation environment, we can obtain a graphical diagram of the configuration patterns for using the first mapping of the matrix multiplication example. It can be seen from Figure 8 that an 8x8 matrix multiplication can be constructed by using two 4x4 structures with one of them a mirror image of the other. In addition, we need to configure the 3rd PE of the mirror image stripe to be an adder.

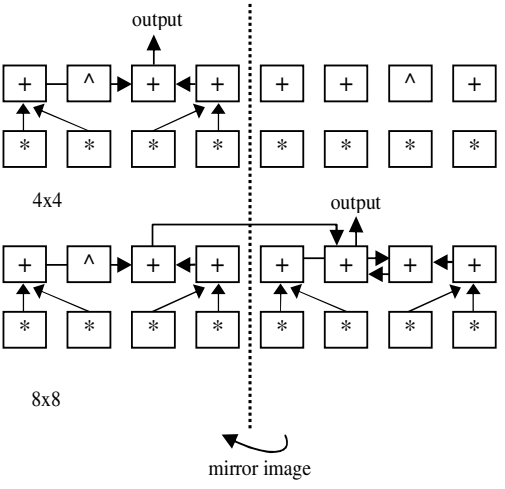


Fig. 8. 4x4 and 8x8 matrix multiplication configuration patterns

Since in our resource model we have restricted the PE connection to laterally 3 of its nearest neighbours, connections between inter-stripes need to obey this restriction. As can be seen, if there is no mirror image and to keep the same type of PEs on both sides, the output connection will tend to shift outwards and would potentially violate the connection restriction. So the actual configuration patterns only require specific interconnections and not the type of PE connections described in the simulation model. This pattern will have a smaller area as in general each PE only needs to route to the nearest PE. The context register can also be smaller now as the multiplexers in the PE needed to select inputs from its neighbours can be reduced.

From our simulation perspective, dynamic reconfiguration can be achieved in two ways. Firstly, we can configure the device with an array of generic PEs (16 nearest routes) and have them 'software' reconfigure to implement different application mappings as demonstrated above. Another alternative is to store the 'fine-tuned' configuration patterns relating to only a particular application and reconfigure the device with these patterns when needed. The latter alternative will need further analysis of the synthesis results for these configuration patterns to obtain the optimum patterns with the least device configuration overhead. We can accomplish this with commercial compilers like Agility in Celoxica's DK Design Suite and Synopsys CoCentric SystemC compiler that can synthesize SystemC directly to high-density FPGA. Therefore, it is important to examine these properties at the system-level.

5 Conclusion and Future work

In this paper, we have presented a simulation environment that is flexible enough to model different computation problems on various known execution models. This produces multiple views/contexts that we can examine for the mapping space, bandwidth, reconfiguration requirements, and configuration patterns of each computational problem. These attributes are imperative to the system scheduling algorithm for the support of dynamic reconfiguration.

From the resulting configuration patterns, we plan to analyse the device configuration overheads for dynamically swapping them into hardware. We will then extend the simulation environment to present the device configuration requirements at the system-level.

References

1. O. Dissel, H. ElGindy, M. Middendorf, H. Schmeck and B. Schmidt, "Dynamic scheduling of tasks on partially reconfigurable FPGAs". In IEE Proceedings on Computers and Digital Techniques, vol. 47, pages 181-188. May 2000.
2. http://www.systemc.org
3. http://www.celoxica.com
4. Grotker. T, "Modeling Software with SystemC 3.0". In 6th European SystemC Users Group Meeting.2002.

5. H. Singh, M.H. Lee. G. Lu, F.J. Kurdahi, N. Bagherzadeh and T. Lang, "MorphoSys: An Integrated Re-configurable Architecture". In Proceedings of the Nato Symposium on System Concepts and Integration, Monterey, CA, April 1998.
6. M. Taylor et al, "The Raw Microprocessor: A Computational Fabric for Software Circuits and General Purpose Programs". IEEE Micro. Mar/Apr 2002.
7. U.J. Kapasi, W.J. Dally, S. Rixner, J.D. Owens and B. Khailany, "The Imagine Stream Processor". In Proceedings IEEE Conference on Computer Design, Sept 16-18, 2002, Freiburg, Germany, pp. 282-288.
8. M. Dyer, C. Plessl, and M. Platzner, "Partially Reconfigurable Cores for Xilinx Virtex". In Proceedings of the Reconfigurable Computing Is Going Mainstream, 12th International Conference on Field-Programmable Logic and Application,.2002
9. Y. Li, T. Callahan, E. Darnell, R. Harr, U. Kurkure and J. Stockwood, "Hardware-Software Co-Design of Embedded Reconfigurable Architectures". IEEE DAC 2000, Los Angeles, California.
10. S. Singh and P.James-Roxby, "Lava and JBits: From HDL to Bitstream in Seconds". In Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines (FCCM), 2001
11. E.L. Horta and J. W. Lockwood, "PARBIT: A tool to transform bitfiles to implement partial reconfiguration of field programmable gate arrays (FPGAs)". Tech. Rep. WUCS-01-13, Washington University in Saint Louis, Department of Computer Science, July 6, 2001.
12. H. Lim, P.Vincenzo, E. Swartzlander, "A Serial-Parallel Architecture for Two-Dimensional Discrete Cosine and Inverse Discrete Cosine Transforms". In IEEE Transactions on Computers, Vol. 49, No. 12, December 2000.
13. H.-W.Lang, R.Maab, M.Schimmler, "The Instruction Systolic Array - Implementation of a Low-Cost Parallel Architecture as Add-On Board for Personal Computers". In Proc. HPCN'94, Lecture Notes in Computer Science 797, Springer-Verlag, 1994

A Switch Wrapper Design for SNA On-Chip-Network

Jiho Chang, Jongsu Yi, and JunSeong Kim

School of Electrical and Electronics Engineering, Chung-Ang University,
221 HeukSeok-Dong DongJak-Gu, Seoul, Korea 156-756
{asura216, xmxm2718}@wm.cau.ac.kr, junkim@cau.ac.kr

Abstract. In this paper we present a design of a switch wrapper as a component of SNA (SoC network architecture), which is an efficient on-chip-network compared to a shared bus architecture in a SoC. The SNA uses crossbar routers to provide the increasing demand on communication bandwidth within a single chip. A switch wrapper for SNA is located between a crossbar router and IPs connecting them together. It carries out a mode of routing to assist crossbar routers and executes protocol conversions to provide compatibility in IP reuse. A switch wrapper consists of a direct router, two AHB-SNP converters, two interface sockets and a controller module. We implement it in VHDL. Using ModelSim simulation, we confirm the functionality of the switch wrapper. We synthesize it using a Xilinx Virtex2 device to determine resource requirements: The switch wrapper seems to occupy appropriate spaces, about 900 gates, considering that a single SNA crossbar router costs about 20,000 gates.

1 Introduction

Enhancement in fabrication technology enables to integrate more IPs in a single chip. In most SoCs design, a bottleneck is the performance of interconnection channels among IPs rather than the performance of IPs including processors. Especially, the performance degrades rapidly when many IPs try to transmit data at the same time. Most of the existing SoCs, which put on a shared bus interconnection architecture, are open to the problem. They tend to use multiplexers instead of tri-state buffers to implement the shared bus architecture. When a master IP broadcasts signals to slaves in the architecture, multiplexers have to drive heavy capacitance, which results in additional power dissipation. As the integration density increases, the lines for an on-chip-bus also increase resulting in heavier propagation delay, higher error rates, and more power dissipation. A new SoC interconnection architecture, which relieves the bottleneck by allowing multiple accesses, would be desirable [3,4].

In fact, SoC design increasingly adopts the concept of NoC(network-on-chip) on-chip communication infrastructure [3]. NoCs have a hierarchical architecture similar to computer networks. Using a data packet-based communication and

T. Srikanthan et al. (Eds.): ACSAC 2005, LNCS 3740, pp. 405–414, 2005.

a switch-based topology NoCs meet the increasing demand of on-chip communication bandwidth. It can scale from a few dozens to several hundreds with a finite communication latency. Also, by providing transparent and efficient on-chip communication NoCs allow to develop hardware resources independently. SNA(SoC Network Architecture) is one of such NoCs to provide multiple channels for on-chip communication by using crossbar routers [2,6].

In this paper, we present a design of a switch wrapper, which is a component of the SNA. Switch wrappers are located between a crossbar router and IPs in SNA. They carry out a mode of routing (so called 'direct routing') to assist crossbar router and execute protocol conversions to provide compatibility in IP reuse. SNA is based on SNP (SoC Network Protocol), an AXI (Advanced eXtensible Interface) compatible on-chip communication protocol, while AMBA AHB (Advanced High-performance Bus) is the dominant on-chip communication protocol in SoC industry. Switch wrappers help to integrate AMBA compliant IPs with SNA. We implement a switch wrapper in VHDL and synthesize it to determine resource requirements.

This paper is organized as follows. Section 2 briefly describes SNA and its operations. In Section 3, we present design tradeoffs and conceptual architecture of the switch wrapper. Section 4 provides experimental results. Finally, Section 5 concludes this paper.

2 SNA (SoC Network Architecture)

As the number of components in a SoC is growing the communication infrastructure within a chip becomes a major concern. Bus-based interconnections have been dominated thanks to the simplicity in design. However, the use of multiple buses may seriously increase the silicon area occupied by deploying unnecessarily large number of lines for various control and data signals. SNP (SoC Network Protocol) is an AXI compatible communication protocol, which requires a less number of wires than a conventional on-chip-bus without significant penalties in performance and design complexity [2]. From the fact that many signals of on-chip-bus are not active at the same time conventional on-chip-bus signals are classified into control, address, and data signals. Then, SNP deploys a set of common wires to transmit the signals in a time-multiplexed way: each transaction in SNP is divided into a request and a response sub-transactions and each sub-transaction consists of one or more phases such as RA (read address), WA (write address), CO (control), RD (read data), WD (write data), RP (response) phases. As a result, SNP reduces the number of interconnection wires without any significant increase in propagation latency [2,7].

SNA is an on-chip-network using SNP [6]. The key idea of SNA is to provide multiple channels concurrently for multiple master IPs requesting accesses to slave IPs. In addition, SNA keeps the interconnection architecture as simple as possible and provides compatibility for AMBA-compliant IPs, which are dominant in the field of SoC industry. SNA can be configured in various ways with the core modules: XR (crossbar router), which consists of a crossbar switch and

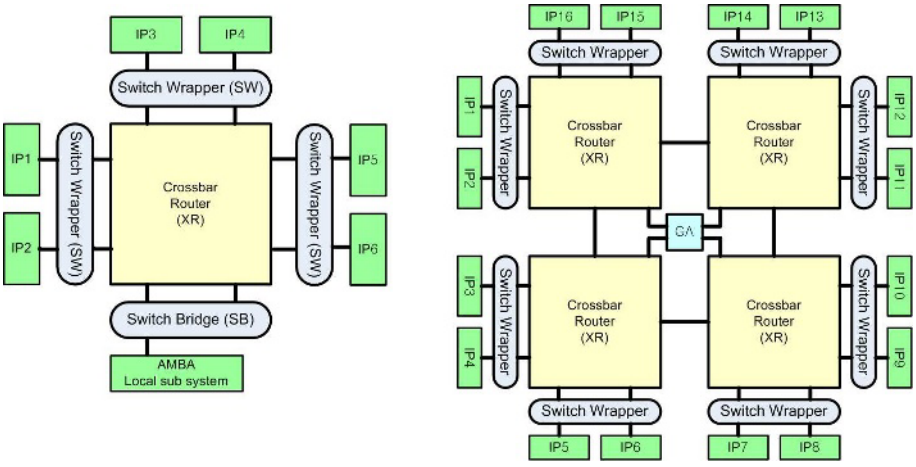


Fig. 1. Two examples of SNA configuration with one XR and four XRs, respectively

a controller, GA (global arbiter), which is only used when multiple XRs are employed, SW (switch wrapper) and SB (switch bridge), which provide consistent interfaces between XRs and IPs. Figure 1 shows possible SNA configurations with one XR and four XRs as examples.

SNA has three operation modes: direct routing, local routing, and global routing [6]. For 'direct routing' a channel is formed between a source and a destination IPs attached on the same SW (or SB). Each SW can support up to 2 IPs and direct routing is used for the communication between the two IPs. The direct routing not only enables a fast channel formation (since no arbitration is necessary there is no extra latency) but also aids XRs to serve other requests. For 'local routing' a single XR makes a channel between a source and a destination IPs. Each XR can support up to 4 SWs, and thus up to 8 IPs, and local routing is used for the communication, which cannot be served by direct routing, among the eight IPs. The local routing costs one-cycle latency. For 'global routing' a GA arranges communication channels across multiple XRs in response to an arbitration signal from an initiating XR, which cannot serve the corresponding interconnection by itself in local routing mode. It reduces the storage and the computation of a GA, and increases the chances of a successful channel formation without further latency. The global routing costs three-cycle latency.

Figure 2 shows performance comparison among various on-chip-networks. The X-axis represents burst transaction length, and the Y-axis is elapsed time to complete transactions. As we expected, the larger the burst length is the longer to complete the transaction regardless of the type of on-chip-networks. For the shared bus based interconnections, only a single transaction can be served at one time. Any conflicts in communication have to be sequentially arranged by putting a certain delay. For the crossbar router based interconnections, more than two transactions can be served simultaneously. This can be further improved by using SWs, which provides faster communication between neighbors.

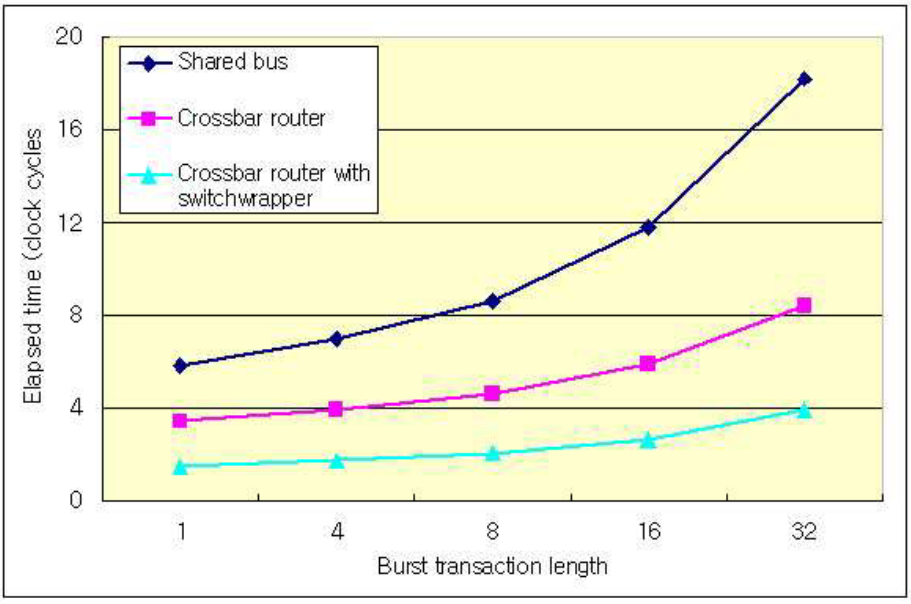


Fig. 2. Performance comparison of various interconnection networks

3 Design of a SNA Switch Wrapper (SW)

A SW connects a XR and two IPs with two IP ports and two XR ports. Each XR port provides SNP interface while each IP port supports either SNP or AHB interface so that all protocol conversions, if necessary, can be carried out within the SW. For direct routing, the two IPs attached on the same SW communicate with each other through the two IP ports. Each SW has information of the attached IPs including identification numbers (IDs) and addresses. Thus, it can detect the destination address upon the request of an IP and directly establish a communication channel to the other IP if the address matches that of the neighbor. A control signal is sent to the attached XR notifying its busy status. For local or global routing, SW delivers signals from IPs to XR or vice versa. When it sends request signals to XR source ID is added to the packet for arbitration. Due to the nature of multi-channel supports in SNA the two IPs on the same SW can communicate at the same time.

Figure 3 shows the conceptual architecture of the SW. It consists of AHB-SNP converter, direct router, interface, and controller modules. Since SNA supports both SNP and AHB compliant IPs, there are various SW operations. The AHB-SNP converter is further divided into master converter and slave converter sub-modules since master and slave IPs need to be served in different way. Also, the direct router is further divided into SNP and AHB router sub-modules to support direct routing between two SNP compliant IPs and two AHB compliant IPs, respectively.

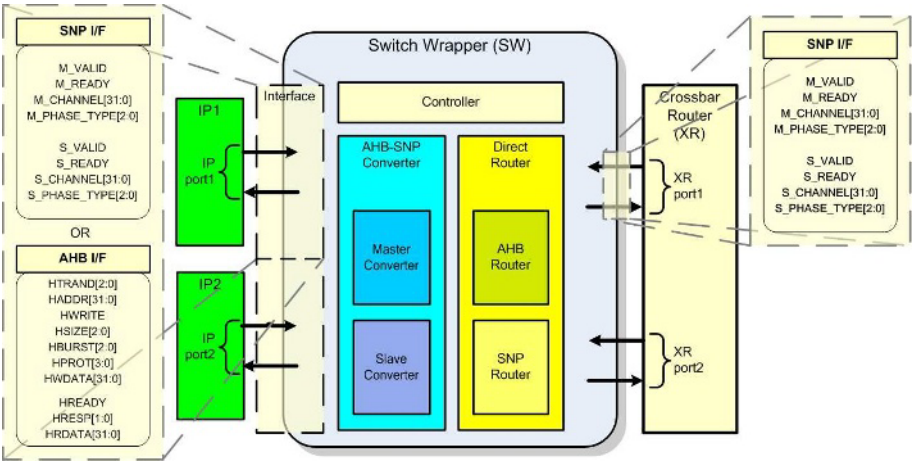


Fig. 3. Overview of a SNA switch wrapper

Table 1 shows all possible configurations of the SW. The column and row represent the type of IPs to serve and each cell contains the required modules to support the corresponding operation. A controller and optional interface modules are common to all operations and omitted from the table. For example, when two SNP compliant master IPs are attached to a SW they will operate in local or global route mode. A controller is the only module to be involved providing signal paths between IPs and XR ("none" in the table). When a SNP compliant master IP and a SNP compliant slave IP are attached to a SW they may operate in direct route mode as well as local and global route modes. A SNP router sub-module needs to be involved in addition to a controller ("1 SNP router" in the

Table 1. Various configurations of a switch wrapper

IP2 \ IP1		SNP		AHB	
		Master	Slave	Master	Slave
SNP	Master	None	1 SNP router	1 Master converter	1 SNP router 1 Slave converter
	Slave	1 SNP router	None	1 SNP router 1 Master converter	1 Slave converter
AMBA	Master	1 Master converter	1 SNP router 1 Master converter	2 Master converters	1 AHB router 1 Master converter 1 Slave converter
	Slave	1 SNP router 1 Slave converter	1 Slave converter	1 AHB router 1 Master converter 1 Slave converter	2 Slave converters

table). An AHB router sub-module is required only when two AHB compliant IPs operate in direct routing mode. AHB-SNP converter module is required when AHB compliant IP communicates with SNP compliant IP in direct routing mode or whenever an AHB compliant IP needs to communicate in local or global routing mode.

Each configuration may be mapped onto a distinct type of SW. In that case, even considering several redundant configurations on the table, there would be tens of distinct SW types. Providing multiple SWs means that SoC designers have to select appropriate type of SWs case-by-case, which is error pron and troublesome. Instead of designing multiple SWs we may design a single generic SW. Since a single SW has to provide all the functionality of the various operations, however, certain sub-modules, which would not be used all the time, need to be included in the generic SW design. There is a trade-offs between easy-of-use and resource usage in the SW design. We choose easy-of-use at the cost of resources hoping a wide spread of SNP in SoC industry.

Figure 4 shows the block diagram of the generic SW design. In order to support both SNP and AHB signals we define local channels to reduce the number of wires within the SW. A local channel consists of two 77 bits unidirectional sub-channels on each direction. The width of a sub-channel comes from the fact that 77 is the maximum number of bits used simultaneously regardless of the IP types in use. The interface module provides a convenient mapping between SNP or AHB interface and a local channel. This is an optional module and an IP can

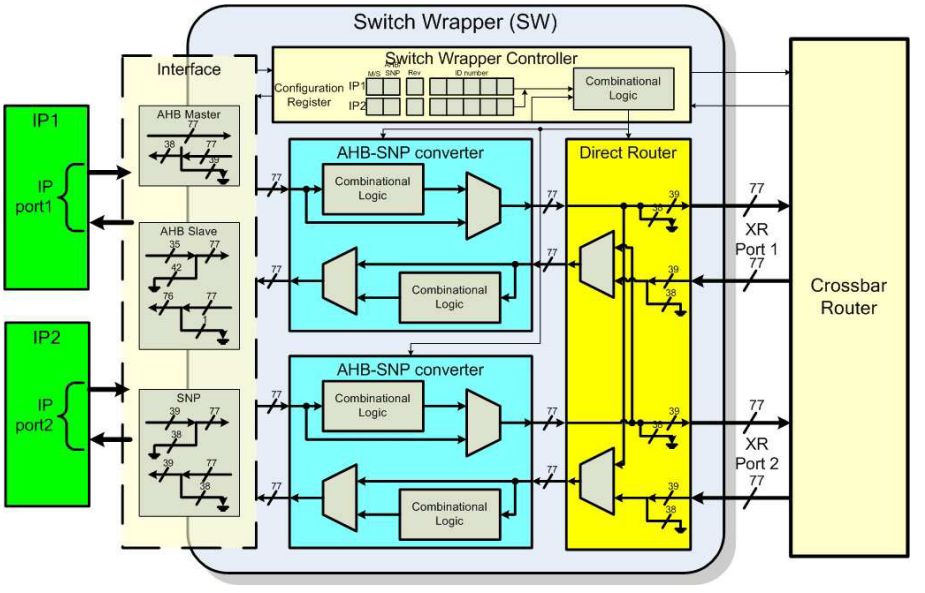


Fig. 4. Block diagram for a generic SNA switch wrapper

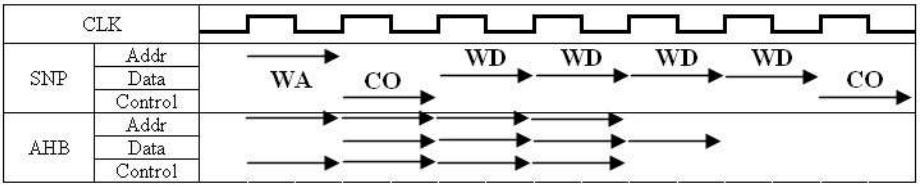


Fig. 5. Comparison of 4-beat incrementing burst write transactions between SNP and AHB communication protocols

be directly connected to the AHB-SNP converter module manually without it. The SW controller has two 8 bit configuration registers to tell the types of the two IPs attached on it. The bit 7 indicates whether an IP is master or slave. The bit 6 indicates whether an IP is SNP compliant or AHB compliant. The bit 5 is reserved for later usage. The bit 4-0 stores the ID information of the attached IP. The value of the configuration registers determines a specific operation of the SW.

One consideration in the generic SW design is the order of the AHB-SNP converter and direct router modules. We place the AHB-SNP converter module in front of the direct router module considering direct routing between AHB compliant IP and SNP compliant IP. In addition, for direct routing between two IPs in the same type the AHB-SNP converter module has a bypass path so that unnecessary conversion can be avoided. In this case, the direct router module carries out either SNP routing or AHB routing depending on the type of IPs in use.

There are a couple of extra considerations in the SNP-AHB converter module design. In AHB protocol data and control signals are transmitted at once while in SNP signals are transmitted throughout one or multiple phases in a time-multiplexing manner. Therefore, the AHB-SNP converter has to either store all AHB signals or delay a completion of an AHB transaction throughout the corresponding SNP phases. For the generic SW we choose the latter by utilizing x_VALID signals in AHB in favor of no storage elements in use.

Figure 5 shows a comparison between SNP and AHB transactions for the same four-beat incrementing burst write transactions. The SNP transactions consist of 1 cycle WA, 1 cycle CO, 4 cycles WD, and 1cycle RP phases in the order. We can see that SNP takes 7 clock cycles to completion while AHB takes 5 clock cycles. AHB has advantage over SNP by 2 clock cycles in latency at the cost of more wires, in this case. Another consideration is that there are no explicit AHB signals, which correspond to the signals in a RP phase of SNP. For the SW design we make the AHB-SNP converter create appropriate signals.

4 Simulation Results

In this section we present a sample simulation result, which confirm the functionality of the SW in various operation modes. The sample test model consists

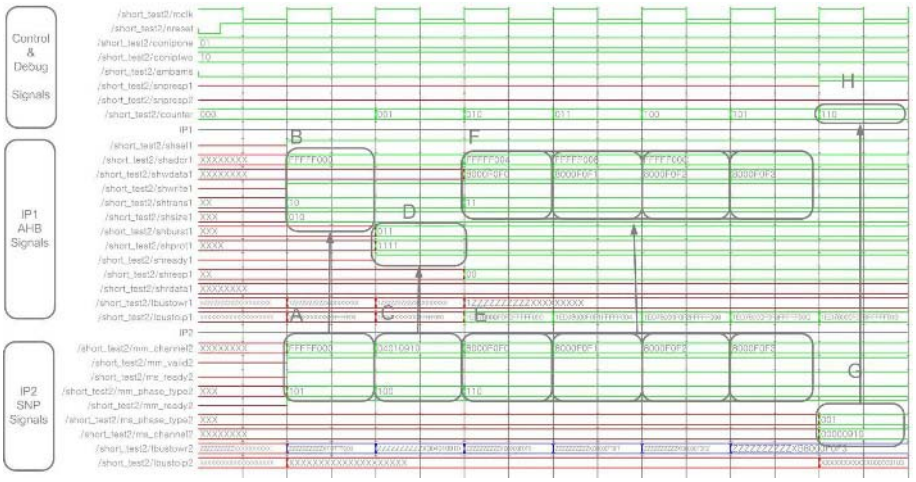


Fig. 6. Direct routing between a SNP compliant master and an AHB compliant slave in four-beat incrementing burst write mode

of a SW with an AHB compliant slave IP1 and a SNP compliant master IP2 communicating each other in direct route mode. Figure 6 shows the procedure that IP2 writes data to IP1 in four-beat incrementing burst write mode:

- In box A, IP2 is in WA phase of SNP transaction sending a base address. You can see the corresponding AHB signals for IP1 after processing SNP2AHB conversion, in box B.
- In box C, a CO phase of SNP transaction is followed in the next cycle. You can see the corresponding AHB signals for IP1, in box D: HBURST='011', HPROT='0011', and HREADY='1', mean that the transaction is a four-beat incrementing burst and privileged data access operation.
- In box E, a WD phase of SNP transaction is followed in the next 4 consecutive cycles. During the WD phase IP2 sends data to IP1 without any extra address signals. You can see the corresponding AHB signals including SW generated address signals for IP1 in box F.
- In box G, a RP phase of SNP transaction is used for sending response, which has no explicit correspondence in AHB transactions.
- In box H, the completion of the whole transaction is notified to the SW so that other transactions, if any, can begin.

Table 2 shows numbers of gates when the generic SNA SW is synthesized using a Xilinx Virtex2 device. A single SW costs either 907 or 1,256 gates without or with the optional interface module, respectively. Considering a single XR consumes about 20,000 gates the generic SW seems to occupy appropriate spaces. Note that there was a trade-offs between easy-of-use and resource usage in the

Table 2. Number of gates for the SNA switch wrapper

SW module	Number of gates
(optional)interface module	0∼329 gates
AHB-SNP converter module	724 gates
direct route module	122 gates
SW controller module	61 gates
Total	907∼1,256 gates

SW design. In this experi-ment, we favor easy-of-use and the size of the SW design can be easily reduced by taking resource usage instead of easy-of-use.

5 Conclusion

In this paper, we present a switch wrapper design as a component of SNA (SoC Network Architecture), which is an efficient on-chip-network using SNP (SoC Network Protocol). A switch wrapper connects a XR (crossbar router) and two IPs with two IP ports and two XR ports. Each XR port provides a SNP interface while each IP port supports either a SNP or an AHB interface. A switch wrapper converts communication protocols between SNP and AHB transactions, if necessary, in order to integrate AHB compliant IPs, which is the dominant in SoC industry, with SNP. A switch wrapper also supports direct routing, which enables a fast channel formation and assists XRs to achieve higher performance than shared bus architecture. Since SNA supports both SNP and AHB compliant IPs, there would be various SW configurations depending on the types of IPs in use. Instead of designing multiple SWs with distinct operations each, we decide to design a single generic SW. It consists of a direct router, two AHB-SNP converters, two interface sockets and a controller module. We implement it in VHDL. Using ModelSim simulation, we confirm the functionality of the SW: with a four-beat incrementing burst write transaction we demonstrate that the SW converts signals between SNP and AHB protocols without loss of extra clock cycles. Also, we find that the number of synthesized logic gates is very small about 900 gates when synthesized using a Xilinx Virtex2 device. Considering that a SNA XR consumes about 20,000 gates the SNA SW design seems to occupy appropriate spaces.

Acknowledgements

This research was partially supported by the MIC(Ministry of Information and Communication), Korea, under the Chung-Ang University HNRC-ITRC (Home Network Research Center) support program supervised by the IITA (Institute of Information Technology Assessment).

References

1. ARM, AMBA Specification Recision 2.0, (1999)
2. Jaesung Lee, Hyuk-Jae Lee, Chanho Lee, SNP: a new communication protocol for SoC, Communications, Circuits and Systems, 2004. ICCCAS 2004. 2004 International Confer-ence on , Volume: 2 , (2004), Pages:1419 - 1423 Vol.2
3. Luca Benini , Giovanni De Micheli, Networks on chips: a new SoC paradigm, Computer , Volume: 35 , Issue: 1 , (2002) Pages:70 - 78
4. Bart Vermeulen, John Dielissen, Kees Goossens and Calin Ciordas, Bringing Communicaion Networks on a Chip: Test and Verification Implications, Communications Magazine, IEEE Volume 41, Issue: 9, Pages:74 - 81, (2003)
5. Pierre Guerrier, Alain Greiner, A generic architecture for on-chip packet-switched intercon-nections, (2000), Proceedings of the conference on Design, automation and test in Europe
6. Sanghun Lee, Chanho Lee, Hyuk-Jae Lee, A new multi-channel on-chip-bus architecture for system-on-chips, SOC Conference, 2004. Proceedings. IEEE International, (2004), Pages:305 - 308
7. The SNP specification, http://capp.snu.ac.kr

A Configuration System Architecture Supporting Bit-Stream Compression for FPGAs

Marco Della Torre[1], Usama Malik[1,2], and Oliver Diessel[1,2]

[1] School of Computer Science and Engineering,
University of New South Wales, Sydney, Australia
[2] Embedded, Real-time, and Operating Systems (ERTOS) Program,
National ICT Australia
{marcodt, umalik, odiessel}@cse.unsw.edu.au

Abstract. This paper presents an investigation and design of an enhanced on-chip configuration memory system that can reduce the time to (re)configure an FPGA. The proposed system accepts configuration data in a compressed form and performs decompression internally. The resulting FPGA can be (re)configured in time proportional to the size of the compressed bit-stream. The compression technique exploits the redundancy present in typical configuration data. An analysis of configurations corresponding to a set of benchmark circuits reveals that data that controls the same types of configurable elements have a common byte that occurs at a significantly higher frequency. This common byte is simply broadcast to all instances of that element. This step is followed by byte updates if required. The new configuration system has modest hardware requirements and was observed to reduce reconfiguration time for the benchmark set by two-thirds on average.

1 Introduction

The high latency of configuration places a significant limitation on the applicability and overall performance of Field Programmable Gate Arrays (FPGAs). This limit is most evident when reconfiguration is performed as part of the overall processing mechanism, such as in dynamically reconfigurable systems. In this paper the background, investigation and design of an enhanced configuration system to reduce this limitation is presented. Our results demonstrate significant performance improvements over currently available devices. The new configuration system reduces the time required to configure an FPGA for typical circuits, requires little additional hardware to that available in current models, and therefore increases the possible applications of FPGAs, while enhancing the performance of systems in which they are already employed.

The technique presented in this paper reduces the (re)configuration time of an FPGA circuit by reducing the amount of configuration data that needs to be loaded onto the device via its configuration port. Data compression is achieved by exploiting the regularities present within typical configurations. An analysis of a set of benchmark circuits from the DSP domain reveals that fragments of

T. Srikanthan et al. (Eds.): ACSAC 2005, LNCS 3740, pp. 415–428, 2005.

configuration data controlling the same type of FPGA resources tend to be similar with at least one byte occurring with a high frequency. This characteristic of typical configurations suggests the following compression technique: partition the configuration data into sets that control the same type of resources in the device; broadcast the most frequent byte in each set on all instances of that resource; then selectively load any bytes that differ from the byte previously broadcast. Using this technique, we observed a two-third reduction in reconfiguration time for the benchmark set.

Techniques to compress FPGA bit-streams have been widely studied. The method in this paper differs mainly in two respects: it shows that a broadcast-based compression technique can be applied on the configurations of a high density FPGA and, in contrast to the previously published methods, it requires significantly less hardware resources to decompress and distribute the data in the configuration memory. These issues are discussed in Section 2. We follow that with an analysis of the configuration data corresponding to typical DSP circuits, which motivates the broadcast-based configuration system presented in Section 4. The proposed model is analysed in Section 5, followed by conclusions and a reference to future work.

2 Related Work and Background

The main focus of this research is on techniques that reduce the reconfiguration time of an FPGA by reducing the amount of configuration data that must be transfered to the memory. This differs from those techniques which compress configuration data in order to reduce the storage requirements and perform decompression before data is loaded into the memory array (e.g. [3]). Apart from compression, architectural techniques such as multi-context FPGAs have also been proposed as a solution to high reconfiguration latency (e.g. [9, 18]). These methods, however, demand significant memory resources. Other proposals, such as pipelined [17] and wormhole [15] reconfiguration are only applicable to specialised FPGA models.

Within the work on compression we identify two categories: the methods in the first category propose special memories that directly accept compressed data. The XC6200 was an earlier FPGA of this type and offered a *wildcard* facility whereby several registers in a column could be written with the same data at once [20]. Luk *et al.* showed that wildcarding can provide near constant-time reconfiguration for highly regular circuits but can be inefficient for irregular cases [10]. Hauck *et al.* presented an algorithm that uses wildcard effectively and reduced reconfiguration time for a set of benchmark circuits to almost a quarter [2].

As FPGAs grew in density, the RAM-style configuration memory had to be compromised since it requires significant hardware resources. The configuration memory in Virtex is implemented as a large number of shift registers that can be individually addressed [19]. Several researchers have investigated compression techniques for this model and achieved 20-85% reduction in bit-stream size for various benchmarks. Dandalis *et al.* studied a dictionary-based compression

technique and found that it demands significant on-chip memory to store the dictionaries [1]. The method presented by Li *et al.* is LZ-based, which has modest on-chip memory requirements but requires a large number of parallel wires across the device [8]. Both of these methods are therefore viewed as impractical for large devices. Recently, Ju *et al.* have described algorithms that exploit both inter- and intra- configuration regularities [14]. However, the required hardware decompressor is not detailed.

Our previous research efforts focused on reducing the amount of configuration data that must be loaded for a circuit by making use of configuration fragments that are already present on-chip [11]. An analysis of a set of benchmark circuits showed that significant *configuration re-use* is possible if the memory allows byte-level access to its registers. However, for a large device, the RAM style, fined-grained access to configuration memory presents with significant address data. Moreover, a RAM style implementation is costly in terms of the wires that are needed to transfer data directly to byte-sized registers.

The above issues were discussed in [12] and a new configuration architecture was presented that allowed byte-level access to configuration memory at a significantly lower cost in terms of the address and wiring overheads. The proposed system implemented a strategy where on-chip data was read into an internal buffer, was modified and finally written back to its destination. The method presented in this paper is an attempt to overcome the increased power consumption incurred due to excessive data movement in the read-modify-write strategy. Moreover, the current memory does not require the user to know the previous configuration state of the FPGA. This architecture is partly inspired by the XC6200's wildcarding mechanism. The main contribution of the present work is that it shows the benefits of such a model, even for high density FPGAs, at negligible additional hardware cost.

3 Empirical Analysis

Compression techniques depend upon the regularities that exist within input data. This section provides an empirical analysis of the frequency distribution of the data within a typical configuration. We first present our assumed device model, a Virtex FPGA [19]. This device was chosen because it is widely used in academia and industry alike. Moreover, Virtex provides a low-level interface to its configuration data, which aids analysis [5].

A Virtex device consists of c columns and r rows of logic and routing resources segmented into so-called configurable logic blocks (CLBs). There are 48 configuration shift-registers per column which span the entire height of the device. Each register configures a portion of a column of the FPGA resources. The data that resides in a register is called a *frame*, which is the smallest unit of configuration. The number of bytes in a frame, f, depends on the number of rows in the device (e.g. for an XCV100, $c = 30, r = 20$ and $f = 56$).

The user supplies the configuration data through an 8-bit wide input port at a configuration clock frequency of at most 66MHz. If the overhead data required due to pipelining of the configuration process is neglected, the time needed to (re)configure the device is directly proportional to the amount of data that is to be transferred to its configuration memory. For an XCV100, a complete configuration consists of 97,652 bytes, which can be loaded in 1.5ms. The configuration delay for the largest member of the family is at least 11.6ms. When an FPGA is reconfigured to implement the various phases of a high-performance, iterative algorithm, e.g. real-time image processing, the size of these overheads can render dynamic reconfiguration infeasible. Low latency reconfiguration techniques are therefore essential to make use of this method.

The process to load configuration data onto a Virtex device uses a DMA approach and works as follows. Load the address of the first frame and the number of consecutive frames that are to be updated. Next, load the required frames onto the device byte by byte. Finally, supply a pad frame in order to flush the internal pipeline. This process needs to be repeated for each block of contiguous frames. The limitations of this addressing model for fine-grained access to the configuration memory were discussed in [12].

Several researchers have considered the problem of re-ordering Virtex frames so as to exploit the similarity between successive frames [8, 14]. We ran several experiments in which various frame orderings were considered. For each ordering, we determined the number and frequency of the unique bytes in the successive frames as this impacts upon any compression technique. Not surprisingly, maximum redundancy in data was observed when the ordering was such that frames at the same offsets within logic columns, which configure the same resources, were considered together. We describe this experiment in detail.

3.1 Experiment 1

Ten common circuits from the DSP domain were considered (Table 1). Table 2 also provides some parameters of the technology-mapped netlists of these circuits indicating their resource requirements. These high-level parameters were used because the CAD tool does not completely report on the low-level utilisation of the device (e.g the number of programmable interconnect points used).

These circuits were mapped onto an XCV100 using ISE5.2 ([4]). This device was chosen because it was the smallest Virtex that could fit all circuits. The total number of 4-input look-up-tables (LUTs) in an XCV100 is 2,400 and the number of bonded IO blocks is 180. Thus, most of the circuits used the available resources sparsely. The circuits were synthesised for minimum area, and the configuration files corresponding to these circuits were generated. These bit files were converted into ASCII for further processing using JBits [5]. In this analysis only 1440 (48×30) frames corresponding to the CLB and switch configurations were considered. The remaining 170 frames in the device correspond to the RAM and IO blocks and are not organised into bundles of 48. In order to simplify our analysis, these were initially ignored.

In the next step, each configuration was partitioned into 48 sets. The i_{th}, $1 \leq i \leq 48$, set consisted of frames that are located in the i_{th} position within each column. We refer to these frames as having the same column offset. Each set, containing 30 frames, was further partitioned into 56 subsets such that the j_{th} subset contained the j_{th} byte from each frame. The size of each of these subsets was thus 30 bytes. The individual bytes occurring in each subset were examined and their frequency within the subset recorded. From this, the average number of unique bytes and their average frequency distribution was determined.

The results are shown in the second column of Table 1. The second column lists the average number of unique bytes at a particular byte position within all frames at the same offset within the CLB columns. The next two columns list the highest and second highest frequencies recorded for individual bytes in these sets. The results show that a single byte value has a frequency of more than 20 on average. In other words, across all frames with the same column offset, at the same byte offset within the frame, just a few bytes values occur on average, and just one of these dominates each set. It was also found that these common bytes differ from row to row. It should be noted that DCT and IIR, the two largest circuits, had much less regularity in their configuration data.

A high level of regularity was observed in the above experiment because frames at the same column offsets configure the same types of resources. By performing similar experiments as above, it was found that frames at different column offsets did not exhibit high degrees of similarity. As a consequence, the architecture outlined in the next section considers frames at the same column offsets as a unit. As for the non-CLB frames, it was found that consecutive frames contain the greatest similarity.

Table 1. Results for Experiments 1 & 2

Circuit	Experiment 1			Experiment 2		
	#Unique bytes	Highest Freq.	2nd Highest Freq.	#Unique bytes	Highest Freq.	2nd Highest Freq.
ammod [4]	3	27	1	7	16	5
bfproc [13]	3	26	1	7	16	5
ccmul [13]	3	27	1	7	16	5
cic3r32 [13]	3	27	1	7	16	5
cosine LUT [4]	4	26	1	7	15	5
dct [4]	6	20	3	11	13	4
ddsynthesiser [13]	2	28	1	6	16	5
dfir [4]	2	28	1	6	16	5
fir_srg [13]	2	28	1	7	16	5
iir [13]	5	22	3	9	15	4

3.2 Experiment 2

The previous experiment attempted to understand byte distributions across the device. This experiment attempts to find regularities vertically within the frames. The objective of this experiment was to determine the average number and distribution of the unique bytes within the frames as it is this similarity that has previously been exploited in [8].

The ten configurations were considered again. Each frame in each configuration was considered. The number of unique bytes in each frame was considered and the frequency table corresponding to each frame was determined. The results are shown in the second column of Table 1, which shows that while regularities exist within the frames, they are not as pronounced as across the frames. This result is also expected as a frame contributes 18 bits, as opposed to some multiple of 8 bits, to each row of resources [19].

3.3 Summary

The results of the above experiments can now be summarised: for typical Virtex configurations, CLB frames at the same column offsets are likely to contain the same data at a particular byte offset with a single common byte occurring with a high frequency within the frames; consecutive non-CLB frames have greatest similarity; and, for the same configurations, intra-frame regularity is significantly less than inter-frame regularity.

4 A New Configuration System Architecture

The proposed scheme is divided into two stages. The first stage configures the non-CLB frames (IOB, BlockRam Interconnect and Centre frames). The second stage configures the CLB frames. During the CLB stage, configuration data is transferred as a block to sets of frames with the same offset within the CLB columns, whereas in the non-CLB stage, the data is transferred as a block to adjacent frames. In the CLB stage, the most common byte is broadcast to every frame in the current set, followed by byte updates to those locations that differ from the broadcast data. The approach followed differs from the current Virtex configuration method, in which adjacent frames are loaded one after another. The system proposed here allows commonality in the blocks of data being sent to frames to be eliminated. This section presents the new configuration system architecture with an XCV100 device in mind. The next section includes a discussion of the scaling of this model to larger devices.

Data is buffered from the byte-wide input port and then transferred to the configuration memory in 30-byte packets. We use the term *byte set* to describe these packets, which are a basic unit of configuration in the proposed design. Although the design described in this section caters for 32 bytes in a packet, our template device only requires the use of 30 at a time.

A byte set is prepared by first supplying the most commonly occurring *beneficiary* byte, which is broadcast to all 30 locations in the byte set. After this, the

user specifies a 4-byte *modification* vector, in which each bit indicates whether a byte in the byte set is to be modified or not. If any bytes are to be modified, the user inputs these in sequence to complete loading the byte set. In XCV100, a byte set can thus be prepared in as few as 5 cycles (1 for the beneficiary byte and 4 for the modification vector) and as many as 34 cycles (29 additional cycles for the non-beneficiary bytes).

Once a byte set is prepared it is distributed throughout the device where the configuration data is shifted into the appropriate frames in parallel. In order to completely configure the selected frames, 56 byte sets, corresponding to the number of bytes within a frame, must be prepared and shifted to the frame registers. These 56 byte sets are subsequently referred to as a *frame set*. For all Virtex family members, 48 frame sets are needed to completely configure the CLB columns and 6 frame sets are needed to configure the non-CLB frames.

Partial configuration is a method for reconfiguring portions of an FPGA instead of the complete device. In Virtex devices, users typically reconfigure one or more vertical bands of the device in order to swap one core and its interconnect for another. The Virtex family supports partial configuration by allowing contiguous frames within a range of frame addresses to be loaded. In this proposal, the user is required to load those frame sets that "touch" the configuration registers spanned by the core that is to be loaded. Usually this will mean all 48 frame sets must be loaded.

To support partial configuration, a couple of mechanisms provide finer control over which parts of the configuration memory are updated and thus over how much data must be loaded. First, the range of frame set addresses that is to be loaded is specified by giving an initial frame set address (FSA_{i}) and a final frame set address (FSA_{F}). Second, prior to loading each frame set, a 4-byte FSA_{we} vector, which *enables writing* to the individual frames within the set, must be loaded. For example, if some incoming core requires that the configuration memory of frames 0–15 and 32–47 in column 12 and frames 12–23 and 36–39 of column 13 be updated, we would load two frame set ranges, the first spanning frame sets 0–23, and the second frame sets 32–47. For frame sets 0–11,

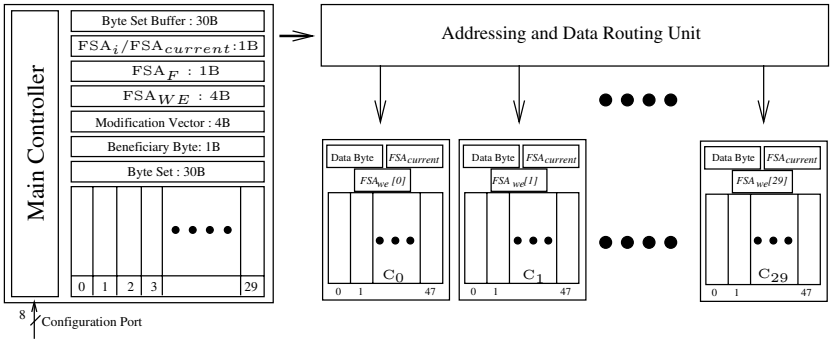


Fig. 1. Configuration memory system overview

all bits of the FSA_{we} vector would be cleared except for the 12th, and for frames sets 12–15 the 13th bit would be asserted as well, then the 12th bit would be cleared for frame sets 16–23, and so on.

4.1 Main Controller

The configuration system we propose consists of two components: a Main Controller and an Addressing and Data Routing Unit (ADRU) (Figure 1). These components are used to organise and distribute the configuration data.

The main controller is the interface between the configuration data input port and the ADRU. The configuration port is assumed to be one byte wide like all Virtex devices. The main controller is responsible for the assembly (decompression) of byte sets prior to their distribution to the configuration memory. The main controller also stores configuration parameters and status information. Configuration parameters include FSA_{i}, FSA_{F} and the FSA_{we} vector, as well as the modification vector. Status information includes the current frame set ad-

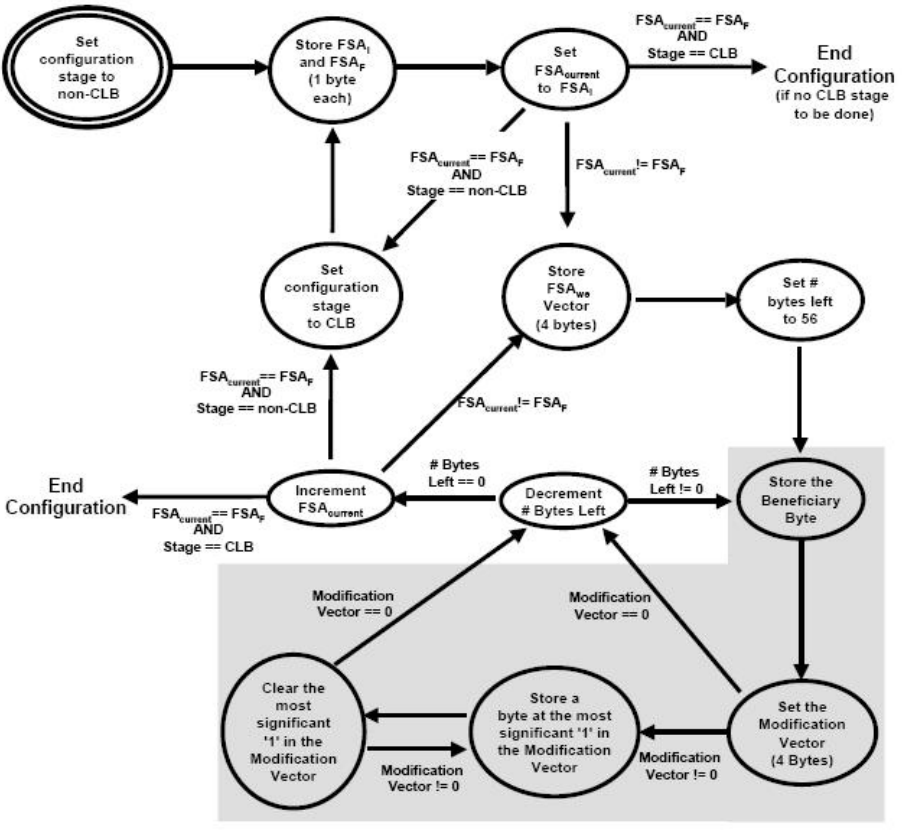


Fig. 2. Main controller state diagram

dress $FSA_{current}$ (the incremented FSA_{i}), the number of byte sets within the frame set remaining to be prepared (between 0 and 56) and the configuration stage (non-CLB or CLB).

The main controller consists of three main sub-components: the *status controller*, a 30-byte *byte set register*, and a *byte set buffer* of equal size. These maintain the status information, the byte set being assembled, and a copy of the previously assembled byte set while it is being broadcast to the configuration memory via the ADRU. In order to assemble a byte set, the beneficiary byte is broadcast to a constant number (8) of byte set registers per cycle while the 4 byte modification vector is being loaded. The controller then loads and routes to the corresponding byte set register entry an additional byte of configuration data for each bit that is set in the modification vector. The overall operation of the main controller is illustrated in the state diagram of Figure 2. The status controller performs the functions described within the unshaded region of the diagram and the byte set register implements the functions within the smaller shaded area.

The control and operation of the byte set buffer, not shown in the state diagram, occurs in parallel with the status controller. While the status controller decrements the number of bytes left in the frame set and checks whether it is equal to zero, the previously prepared byte set is transferred from the byte set register to the byte set buffer. While the operation of the main controller continues to prepare the next byte set the byte set buffer is free to transfer the previous byte set to the configuration memory of the device using the ADRU.

4.2 Addressing and Data Routing Unit

The ADRU is responsible for transferring configuration data from the byte set buffer to the configuration memory elements on the device (Figure 3). Since a

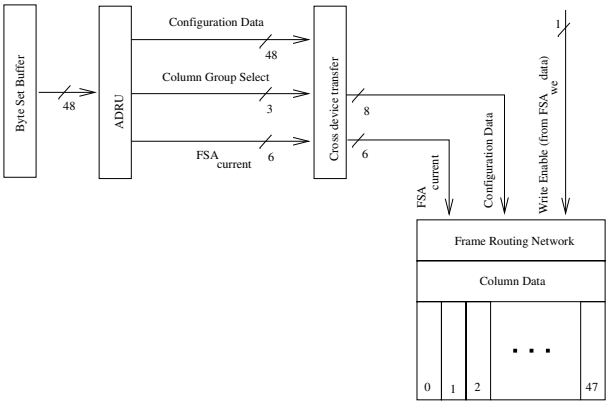


Fig. 3. Address and data routing unit arrangement. Note: just one column is depicted in this diagram.

byte set can be prepared in as few as 5 cycles, the ADRU must transfer 30 bytes to the configuration memory within this period. In order to minimise the bus width of the ADRU, the proposal envisages transferring 6 bytes per cycle. A 3-bit column group select signal indicates which 6-byte fragment is currently being transferred. When the byte set register unit is accepting the beneficiary byte of the next byte set, the ADRU copies the first 6-byte section of the byte set buffer to the device. The ADRU uses the configuration stage, $FSA_{current}$ and column group select signals to determine how the data is routed to the device. The FSA_{we} vector selects which bytes are actually written to the device, thus disabling any frame registers not being configured.

5 Analysis of the System

Hardware Requirements

The hardware requirements of the proposed configuration system are modest and comparable with those currently present in the Virtex family. The main change is to have a somewhat wider data distribution network in the ADRU — up to 8 bytes of data in parallel (when fully expanded, as outlined below), compared with 4 bytes — and the ability to shift data into 8 frame registers in parallel (when expanded — for XCV100, 6 frames are targeted). Initial modelling of the area and power needs of our design using Design Compiler from Synopsis suggest power consumption will increase by a factor of about 1.5 over the current Virtex system during configuration, but will be compensated for by having the configuration period reduced by a factor of 2 to 3. This estimate is based on estimates carried out on the system described in [12], which proposes two buses rather than one, and accessed each configuration register twice per cycle in order to be able to *modify* the configuration currently on chip.

The timing of the modification vector update unit was considered to be the critical element within the controller design. This unit employs successive bit-clearing logic to route non-beneficiary bytes to the byte set register and to test for the need to load further byte set data. This logic is described by the left pair of states in the shaded region of the state diagram depicted in Figure 2. Our implementation of the successive bit-clearing logic uses a chain of two-input XOR gates, 2 per modification vector bit, and thus has a a delay proportional to the size of the modification vector. With current 90nm process technology, the delay for a 32-bit vector was found to be 1.6ns and is thus insignificant.

To gauge the delay of the ADRU we assumed the propagation of data across the device could be supported at the 66MHz configuration clock speed currently used by Virtex. Should this not be possible, the data transfer could easily be pipelined.

Benchmark Performance

In order to evaluate the performance of our proposed configuration system we compared the amount of data currently needed to configure the circuits described

Table 2. Bit-stream sizes in bytes (including overheads) and percentage reduction in bit-stream size for benchmark circuits using the proposed scheme (Modification Vector) and an alternative (Random Access)

Circuit	#LUTs	#IOBs	#Nets	Modification Vector Bit-stream		Random Access Bit-stream	
				Size	% Red	Size	% Red
ammod [4]	271	45	990	28,412	70.9	30,960	68.3
bfproc [13]	418	90	1,347	30,229	69.0	34,140	65.0
ccmul [13]	262	58	905	27,179	72.2	28,490	70.8
cic3r32 [13]	152	42	736	26,667	72.7	27,450	71.9
cosine LUT [4]	547	45	2,574	31,710	67.5	37,580	61.5
dct [4]	1,064	78	5,327	54,315	44.4	71,394	26.9
ddsynthesiser [13]	70	44	759	25,704	73.7	25,214	74.2
dfir [4]	179	43	782	26,262	73.1	26,668	72.7
fir_srg [13]	216	16	726	26,143	73.2	26,480	72.9
iir [13]	894	62	2,907	42,011	57.0	57,108	41.5
Average				32,079	67.2	36,548	62.6

in Section 3 on a Virtex XCV100 with the amount of data needed using our scheme. This data corresponds directly to the number of configuration clock cycles needed to load the configuration bit-stream and to configure the device. Refer to Table 2.

The third and fourth column from the right in this table list the total number of bytes, including overheads, that are needed with our proposal and the percentage reduction in bit-stream size. On average, just over 32,000 bytes are needed to configure each circuit while 97,652 bytes are needed using the current XCV100 configuration interface. This is largely due to the way the circuits were mapped, since each required a complete configuration of the device. Primarily this was due to nets crossing the entire device. It would perhaps be more fair to compare the methods when the circuits are compacted into as few columns as possible. Nevertheless, the results are encouraging for most circuits, with an average reduction in configuration bit-stream size and latency of 67.2%.

We are concerned that the results for the DCT and IIR circuits indicate less regular, larger circuits will cause significant loss of benefit from compression, particularly as the device utilisation approaches 100%. We are currently investigating this effect with high-stress circuits.

Random Access Byte Set Modification

The proposed configuration system has a relatively high fixed overhead of 12,316 bytes for a complete configuration. This overhead, comprising byte set modification vectors, frame write masks and frame set ranges, may be too high for partial reconfiguration. To partially configure a single column of the device in

which all 48 frames are touched results in 48 frame sets having to be written with an overhead of 10,754 bytes.

We therefore examined the performance of an alternative scheme in which non-beneficiary bytes are addressed using a 1-byte address for each byte to be modified in the byte set. This approach should benefit byte sets in which the number of bytes that differ from the broadcast byte is less than four. For the complete configurations under test, we found that this typically led to a net increase in bit-stream size. See the right pair of columns in Table 2.

It is expected that this method will have a benefit when the number of non-beneficiary bytes is less than 4 on average. This is more likely for updates covering a small number of columns but will be less likely as the utilisation or the functional density of the frames covered increases. Another factor to be considered with this alternative is that a byte set could be ready for distribution every 2 cycles: just 1 beneficiary byte and 1 end of byte set marker may suffice to specify a complete byte set of 30 bytes. With the XCV100 device, up to 15 frames would therefore need to be configured per cycle in order to maintain the configuration bandwidth at the input port.

Scalability

The proposed configuration method could be adapted for use in larger devices by repeating and/or expanding the design. The number of frames in each column is fixed for all Virtex series devices, and so need not be considered. Similarly, the number of non-CLB columns in Virtex series devices is fixed. However, there are 96 columns of CLBs in the largest (XCV1000) device, and each frame contains 156 bytes. This represents a significant increase over the XCV100 device in the amount of data to be transferred. The increase in the number of bytes per frame only affects the size of the counter controlling the current byte position, increasing it from 6 bits (56 bytes per frame) to 8 bits (156 bytes per frame). However, the large increase in the number of CLB columns needs further consideration.

Repetition refers to adding one CLB configuration stage for each additional set of 32 CLB columns. This strategy necessitates that the controller keep track of the current CLB configuration stage. For example, in the XCV1000 there would be 1 non-CLB stage and 3 CLB stages. The CLB configuration stage is broadcast along with the configuration data in order to configure the correct subset of columns. If necessary, the data bus would be pipelined to cope with delays in broadcasting the configuration and control data across the chip.

Expansion refers to the enlargement of existing structures to avoid the use of multiple CLB configuration stages and the need to transfer additional configuration stage data. The Virtex XCV1000 could be implemented using a byte set size of 96. The modification vector system would then have a latency of approximately 96 gate delays. In 90nm process technology this critical path length allows a configuration clock frequency of approximately 200MHz, which at more than twice the speed of current Virtex devices, is adequate.

The broadcast of the beneficiary byte does not pose a problem in an expanded system since at most just 8 bytes must be written to per cycle. The

number of bytes needing to be transferred from the byte set buffer to the device would be adapted to 8 bytes/cycle for large devices and therefore does not add prohibitively to the hardware requirements. Accordingly, the number of FSA_{we} bits needing to be broadcast increases from 6 to 8. Since each of these 8 bytes could be written to any of 12 sets of contiguous columns, the size of the column select group increases to 4 bits. The 200% increase from the XCV100 to the XCV1000 in the number of configuration columns thus necessitates an expansion of the data bus width from 57 to 76 bits. Indeed, if the configuration data is ignored, the overhead in addressing data increases from 9 to just 12 bits.

Details on the configuration architecture employed in the latest Virtex-4 series of FPGAs offered by Xilinx are vague. We understand these devices may be thought of as a small, vertically aligned stack of enlarged Virtex-1 devices. The configuration memory is thus partitioned into a small number of wide horizontal bands or pages corresponding to the smaller units comprising the stack, and Virtex-4 frames are partitioned into a small number of sub-units that are individually addressable. We see our approach as being applied at this sub-unit level, with a shared or separate controller for each page of sub-frames.

6 Conclusions and Future Work

This paper has presented an analysis of configurations corresponding to common DSP circuits on a Virtex FPGA. It was found that frames at the same column offsets are likely to contain the same data with one byte occurring with a high frequency at the same byte offset within the frames. A new configuration system was developed to exploit this phenomenon. The architecture simply broadcasts the most frequent byte on selected frames followed by updates to individual bytes where needed. The new design reduced the (re)configuration time for the benchmark set by two-thirds with modest hardware additions.

In the future, we would like to extend our method to include configuration caching (studied by many researchers e.g. [7, 16, 6]). We are currently investigating the possibility of caching the update bytes in our method. This is likely to further reduce the (re)configuration time especially for dense circuits.

Acknowledgements. National ICT Australia is funded through the Australian Government's *Backing Australia's Ability* initiative, in part through the Australian Research Council.

References

1. A. Dandalis and V. Prasanna. Configuration compression for FPGA-based embedded systems. *ACM International Symposium on Field-Programmable Gate Arrays*, pages 187–195, 2001.
2. S. Hauck, Z. Li, and E. Schwabe. Configuration compression for the Xilinx XC6200 FPGA. *IEEE Transactions on Computer Aided Design on Integrated, Circuits and Systems, Volume 18 Number 8*, pages 1237–1248, 1999.

3. M. Huebner, M. Ullmann, F. Weissel, and J. Becker. Real-time configuration code decompression for dynamic FPGA self-reconfiguration. *Reconfigurable Architectures Workshop*, 2004.
4. ISE Version 5.2. *Xilinx Inc.*, 2002.
5. JBits SDK. *Xilinx Inc.*, 2000.
6. I. Kennedy. Exploiting redundancy to speedup reconfiguration of an FPGA. *Field Programmable Logic*, pages 262–271, 2003.
7. Z. Li, K. Compton, and S. Hauck. Configuration cache management techniques for FPGAs. *IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 22–36, 2000.
8. Z. Li and S. Hauck. Configuration compression for Virtex FPGAs. *IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 2–36, 2001.
9. X. Ling and H. Amano. WASMII: A data driven computer on a virtual hardware. *IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 33–42, 1993.
10. W. Luk, N. Shirazi, and P. Cheung. Compilation tools for run-time reconfigurable designs. *IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 56–65, 1997.
11. U. Malik and O. Diessel. On the placement and granularity of FPGA configurations. *International Conference on Field-Programmable Technology*, pages 161–168, 2004.
12. U. Malik and O. Diessel. A configuration memory architecture for fast Run-Time-Reconfiguration of FPGAs. *International Conference on Field Programmable Logic*, 2005.
13. U. Meyer-Baese. Digital signal processing with Field Programmable Gate Arrays. *Springer*, 2001.
14. J. Pan, T. Mitra, and W. Wong. Configuration bitstream compression for dynamically reconfigurable FPGAs. *International Conference on Computer Aided Design*, pages 766–773, 2004.
15. A. Ray and P. Athanas. Wormhole run-time reconfiguration. *International Symposium on Field-Programmable Gate Arrays*, pages 79–85, 1997.
16. S. Sadhir, S. Nath, and S. Goldstein. Configuration caching and swapping. *Field-Programmable Logic and Applications*, pages 192–202, 2001.
17. H. Schmit. Incremental reconfiguration for pipelined applications. *IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 47–55, 1997.
18. S. Trimberger. A Time-Multiplexed FPGA. *IEEE Symposium on FPGA-Programmable Custom Computing Machines*, pages 22–28, 1997.
19. Virtex 2.5V Field Programmable Gate Arrays Data Sheet, Version 1.3. *Xilinx, Inc.*, 2000.
20. XC6200 Field Programmable Gate Arrays, version 1.10. *Xilinx, Inc.*, 1997.

Biological Sequence Analysis with Hidden Markov Models on an FPGA

Jacop Yanto, Timothy F. Oliver, Bertil Schmidt, and Douglas L. Maskell

School of Computer Engineering, Nanyang Technological University,
Singapore 639798
{yanto_jakop, tim.oliver}@pmail.ntu.edu.sg,
{asbschmidt, asdouglas}@ntu.edu.sg

Abstract. Molecular biologists use Hidden Markov Models (HMMs) as a popular tool to statistically describe protein families. This statistical description can then be used for sensitive and selective database scanning, e.g. new protein sequences are compared with a set of HMMs to detect functional similarities. Even though efficient dynamic programming algorithms exist for the problem, the required scanning time is still very high, and because of the rapid database growth finding fast solutions is of high importance to research in this area. In this paper we present how reconfigurable architectures can be used to derive an efficient fine-grained parallelization of the dynamic programming calculation. It is described how this technique leads to significant runtime savings for HMM database scanning on a standard off-the-shelf FPGA.

1 Introduction

Scanning sequence databases is a common and often repeated task in molecular biology. The need for speeding up this treatment comes from the recent developments in genome-sequencing projects, which are generating an enormous amount of data. This has resulted in a very rapid growth of the biosequence banks. The scan operation consists of finding similarities between a particular query sequence and all the sequences of a bank. This operation allows biologists to point out sequences sharing common subsequences. From a biological point of view, it leads to identify similar functionality.

However, identifying distantly related homologs is still a difficult problem. Because of sparse sequence similarity, commonly used comparison algorithms like BLAST [1] or Smith-Waterman [23] often fail to recognize their homology. Since HMMs provide a position-specific description of protein families, they have become a powerful tool for high sensitivity database scanning. HMMs can find that a new protein sequence belongs to the modeled family, even with low sequence identity [7].

An HMM is compared with a subject sequence by dynamic programming (DP) based alignment algorithms, such as Viterbi or Expectation Maximization, whose complexities are quadratic with respect to the sequence and model length. There have been basically two methods of parallelizing HMM database scanning: one is based on the parallelization of the DP algorithm, the other is based on the distribution of the

T. Srikanthan et al. (Eds.): ACSAC 2005, LNCS 3740, pp. 429 – 439, 2005.

computation of pairwise comparisons. Fine-grained parallel architectures, like linear SIMD arrays and systolic arrays, have been proven as a good candidate structure for the first approach [4,12,22], while more coarse-grained networks of workstations are suitable architectures for the second [8,15].

Special-purpose systolic arrays provide the best area/performance ratio by means of running a particular algorithm [14]. Their disadvantage is the lack of flexibility with respect to the implementation of different algorithms. Several massively parallel SIMD architectures have been developed in order to combine the speed and simplicity of systolic arrays with flexible programmability [3,5,20]. However, because of the high production costs involved, there are many cases where announced second-generation architectures have not been produced. The strategy to high performance sequence analysis used in this paper is based on FPGAs. FPGAs provide a flexible platform for fine-grained parallel computing based on reconfigurable hardware. Since there is a large overall FPGA market, this approach has a relatively small price/unit and also facilitates upgrading to FPGAs based on state-of-the-art technology. We will show how this leads to a high-speed implementation on a Virtex II XC2V6000. The implementation is also portable to other FPGAs.

This paper is organised as follows. In Section 2, we introduce the Viterbi algorithm used to align a profile HMM to a sequence. Section 3 highlights previous work on parallel architectures for biological sequence analysis. The parallel algorithm and its mapping onto a reconfigurable platform are explained in Section 4. The performance is evaluated and compared to previous implementations in Section 5. Section 6 concludes the paper with an outlook to further research topics.. The preparation of manuscripts which are to be reproduced by photo-offset requires special care. Papers submitted in a technically unsuitable form will be returned for retyping, or canceled if the volume cannot otherwise be finished on time.

2 Viterbi Algorithm

Biologists have characterized a growing resource of protein families that share common function and evolutionary ancestry. Hidden Markov models (HMMs) have been identified as a suitable mathematical tool to statistically describe such families. Consequently, databases of HMMs for protein families have been created [2]. HMMs have become a powerful tool for high sensitivity database scanning, because they can provide a position-specific description of protein families. HMMs can identify that a new protein sequence belongs to the modeled family, even with low sequence identity [7]. A protein sequence can be aligned to a HMM to determine the probability if it belongs to the modeled family. This alignment can be computed by a DP-based alignment algorithm: the Viterbi algorithm.

The structure of an HMM to model a protein sequence family is called a *profile HMM* (see Fig. 1) [6]. It consists of a linear sequence of nodes. Each node has a match (*M*), insert (*I*) and delete state (*D*). Between the nodes are transitions with associated probabilities. Each match state and insert state also contains a position-specific table with probabilities for emitting a particular amino acid. Both transition and emission probabilities can be generated from a multiple sequence alignment of a protein family.

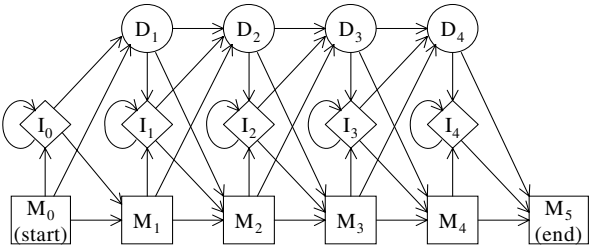


Fig. 1. The transition structure of a profile HMM of length 4. Squares represent match states, circles represent delete states and diamonds represent insertions.

An HMM can be compared (aligned) with a given sequence to determine the probability that the sequence belongs to the modeled family. The most probable path through the HMM that generates a sequence equal to the given sequence determines the similarity score. The well-known Viterbi algorithm computes this score by DP. The computation is given by the following recurrence relations.

$$M(i,j) = e(M_j, s_i) + \max\begin{cases} M(i-1,j-1) + tr(M_{j-1}, M_j) \\ I(i-1,j-1) + tr(I_{j-1}, M_j) \\ D(i-1,j-1) + tr(D_{j-1}, M_j) \end{cases}$$

$$I(i,j) = e(I_j, s_i) + \max\begin{cases} M(i-1,j) + tr(M_j, I_j) \\ I(i-1,j) + tr(I_j, I_j) \\ D(i-1,j) + tr(D_j, I_j) \end{cases}$$

$$D(i,j) = \max\begin{cases} M(i,j-1) + tr(M_{j-1}, D_j) \\ I(i,j-1) + tr(I_{j-1}, D_j) \\ D(i,j-1) + tr(D_{j-1}, D_j) \end{cases}$$

where $tr(state1,state2)$ is the transition cost from *state1* to *state2* and $e(M_j,s_i)$ is the emission cost of amino acid s_i at state M_j. $M(i,j)$ denotes the score of the best path matching subsequence $s_1 \ldots s_i$ to the submodel up to state j, ending with s_i being emitted by state M_j. Similarly $I(i,j)$ is the score of the best path ending in s_i being emitted by I_j, and, $D(i,j)$ for the best path ending in state D_j. Initialization and termination are given by $M(0,0)=0$ and $M(n+1,m+1)$ for a sequence of length n and an HMM of length m. By adding jump-in/out costs, null model transitions and null model emission costs the equation can easily be extended to implement Viterbi local scoring (see e.g. [6]).

Example of a global aligning of a sequence to an HMM is illustrated in Figures 2, 3, and 4. A profile HMM of length 4 transition scores is shown in Figure 2. The emission scores of the *M*-states are shown in Figure 3. The emission scores of all *I*-states are all set to zero, i.e. $e(I_j,s_i) = 0$ for all i ,j. The Viterbi DP matrix for computing the global alignment score of the protein sequence `HEIKQ` and the given HMM is shown in Figure 4. The three values *M*, *I*, *D* at each position are displayed as ${}_{\mathbf{D}}\mathbf{M}^{\mathbf{I}}$. A traceback procedure starting at $M(6,5)$ and ending at $M(0,0)$ (shaded cells in Figure 4) delivers the optimal path through the given HMM emitting the sequence `HEIKQ`.

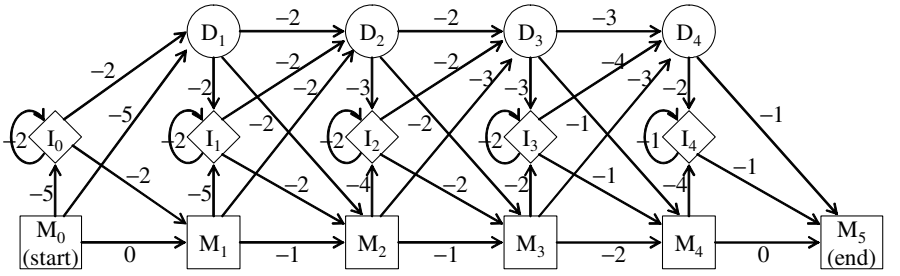


Fig. 2. The given profile HMM of length 4 with transition scores

	A	C	D	E	F	G	H	I	K	L	M	N	P	Q	R	S	T	V	W	Y
M_1	−1	−1	−1	−1	1	−1	3	1	−1	−1	−1	−1	−1	−1	−1	−1	−1	−1	−1	−1
M_2	1	0	0	1	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	1
M_3	2	0	0	0	0	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0
M_4	−1	−1	1	−1	−1	−1	1	−1	2	−1	−1	−1	−1	−1	−1	−1	1	−1	−1	−1

Fig. 3. Emission scores of the *M*-states for the HMM in Figure 2

	0	1	2	3	4	5
∅	$*0^{*}$	$_{-5}-\infty^{-\infty}$	$_{-7}-\infty^{-\infty}$	$_{-9}-\infty^{-\infty}$	$_{-12}-\infty^{-\infty}$	
H	$_{-\infty}-\infty^{-5}$	$_{-7}3^{-7}$	$_{1}-7^{-9}$	$_{-1}-9^{-12}$	$_{-4}-9^{-14}$	
E	$_{-\infty}-\infty^{-7}$	$_{-9}-8^{-2}$	$_{-4}3^{-2}$	$_{0}-1^{-4}$	$_{-3}-3^{-6}$	
I	$_{-\infty}-\infty^{-9}$	$_{-11}-6^{-4}$	$_{-6}-4^{-1}$	$_{-3}2^{-3}$	$_{-1}-2^{-5}$	
K	$_{-\infty}-\infty^{-11}$	$_{-13}-10^{-6}$	$_{-8}-5^{-3}$	$_{-5}-3^{0}$	$_{-4}2^{-3}$	
Q	$_{-\infty}-\infty^{-13}$	$_{-15}-12^{-8}$	$_{-10}-8^{-5}$	$_{-7}-5^{-2}$	$_{-6}-2^{-2}$	
						−2

Fig. 4. The Viterbi DP matrix for computing the global alignment score of the protein sequence HEIKQ and the given HMM

3 Related Work

A number of parallel architectures have been developed for biological sequence analysis (mainly for the Smith-Waterman algorithm [23]). In addition to architectures specifically designed for sequence analysis, existing programmable sequential and parallel architectures have been used for solving sequence alignment problems. Special-purpose hardware implementations can provide the fastest means of running a particular algorithm with very high PE density. However, they are limited to one single algorithm, and thus cannot supply the flexibility necessary to run a variety of algorithms required analyzing DNA, RNA, and proteins. P-NAC was the first such machine and computed edit distance over a four-character alphabet [16]. More recent examples, better tuned to the needs of computational biology, include BioScan, BISP, and SAMBA [4,12,22].

An approach presented in [20] is based on instruction systolic arrays (ISAs). ISAs combine the speed and simplicity of systolic arrays with flexible programmability. Several other approaches are based on the SIMD concept, e.g. MGAP [3] and Kestrel [5]. SIMD and ISA architectures are programmable and can be used for a wider range of applications, such as image processing and scientific computing. Since these architectures contain more general-purpose parallel processors, their PE density is less than the density of special-purpose ASICs. Nevertheless, SIMD solutions can still achieve significant runtime savings. However, the costs involved in designing and producing SIMD architectures are quite high. As a consequence, none of the above solutions has a successor generation, making upgrading impossible.

Reconfigurable systems are based on programmable logic such as field-programmable gate arrays (FPGAs) or custom-designed arrays. They are generally slower and have lower PE densities than special-purpose architectures. They are flexible, but the configuration must be changed for each algorithm, which is generally more complicated than writing new code for an instruction set architecture (ISA) architecture. Several solutions including Splash-2 [13] and Decipher [24] are based on FPGAs while PIM has its own reconfigurable design [9]. Solutions based on FPGAs have the additional advantage that they can be regularly upgraded to state-of-the-art technology. This makes FPGAs a very attractive alternative to special-purpose and SIMD architectures.

Previously published work on using FPGAs for biological sequence analysis has mainly focused on implementations of the Smith-Waterman algorithm [19,25,26], BLAST [17], and multiple sequence alignment [18]. In this paper we present how the Viterbi algorithm can be efficiently mapped onto reconfigurable hardware.

4 Mapping onto a Reconfigurable Platform

The three values of I, D, and M of any cell in the Viterbi DP matrices can only be computed if the values of all cells to the left and above have been computed. But the calculations of the values of diagonally arranged cells parallel to the minor diagonal are independent and can be done simultaneously. Assuming we want to compare/align a subject sequence to a query model (profile HMM) on a linear array of processing elements (*PEs*) this parallelization is achieved by mapping the Viterbi calculation as follows: one PE is assigned to each node of the query model. The subject sequence is then shifted through the linear chain of PEs from left to right (see Figure 5). During each step, one elementary matrix computation is synchronously performed in each PE. If $l1$ is the length of the subject sequence and $l2$ is the length of the query string/model, the comparison is performed in $l1+l2-1$ steps on $l1$ PEs, instead of $l1 \times l2$ steps required on a sequential processor.

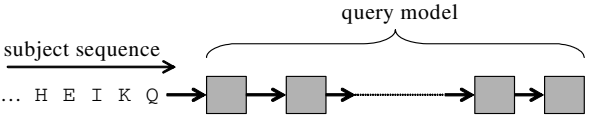


Fig. 5. Systolic sequence comparison on a linear processor array

Figure 6 shows our design for each individual PE. It contains local memory to store the following temporary DP matrix values:

1. *Diagonal*: $M(i-1,j-1)$, $I(i-1,j-1)$, $D(i-1,j-1)$ in *m_diag_d*, *i_diag_d*, *d_diag_d*
2. *Above*: $M(i-1,j)$, $I(i-1,j)$, $D(i-1,j)$ in *m_above_d*, *i_above_d*, *d_above_d*
3. *Left*: $M(i,j-1)$, $I(i,j-1)$, $D(i,j-1)$ in *m_left_d*, *i_left_d*, *d_left_d*

The PE holds the emission probabilities $e(M_j,s_i)$ and $e(I_j,s_i)$ for the corresponding HMM state in two LUTs. The look-ups of $e(M_j,s_i)$ and $e(I_j,s_i)$ and their addition to *max_diag_d* and *max_left_d* are done in one clock cycle. The results (*m_out*, *i_out*, *d_out*) are then passed to the next PE in the array together with the sequence character (*s_out*). All additions are performed using saturation arithmetic.

Assuming, we are aligning the subject sequence $S = s_1 \ldots s_M$ of length M to a query HMM of length K on a linear processor array of size K using the Viterbi algorithm. As a preprocessing step, the transition and emission probabilities of states M_j, I_j, and D_j are loaded into PE j, $1 \le i \le K$. S is then completely shifted through the array in $M+K-1$ steps as displayed in Figure 5. In iteration step k, $1 \le k \le M+K-1$, the values $M(i,j)$, $I(i,j)$, and $D(i,j)$ for all i, j with $1 \le i \le M$, $1 \le j \le K$ and $k=i+j-1$ are computed in parallel in all PEs, $1 \le j \le K$, within a single clock cycle. For this calculation PE j, $2 \le j \le K$, receives the values $M(i,j-1)$, $I(i,j-1)$, $D(i,j-1)$ and s_i from its left neighbor $j-1$, while the values $M(i-1,j-1)$, $I(i-1,j-1)$, $D(i-1,j-1)$, $M(i-1,j)$, $I(i-1,j)$, $D(i-1,j)$, s_i, $e(M_j,s_i)$, $e(I_j,s_i)$, $tr(M_{j-1},M_j)$, $tr(I_{j-1},M_j)$, $tr(D_{j-1},M_j)$, $tr(M_j,I_j)$, $tr(I_j,I_j)$, $tr(D_j,I_j)$, $tr(M_{j-1},D_j)$, $tr(I_{j-1},D_j)$, $tr(D_{j-1},D_j)$ are stored locally.

Thus, it takes $M+K-1$ steps to compute the alignment score with the Viterbi algorithm. However, notice that after the last character of S enters the array, the first character of a new subject sequence can be input for the next iteration step. Thus, all subject sequences of the database can be pipelined with only one-step delay between two different sequences.

So far we have assumed a processor array equal in size of the query model length. In practice, this rarely happens. Since the length of the HMMs may vary, the computation must be partitioned on the fixed size processor array. The query model is usually larger than the processor array. For sake of clarity we firstly assume a query sequence of length K and a processor array of size N where K is a multiple of N, i.e. $K=p \cdot N$ where $p \ge 1$ is an integer. A possible solution is to split the computation into p passes:

The first N states of the query model are assigned to the processor array and the corresponding substitution table columns loaded. The entire database of subject sequences to be aligned to the query model then crosses the array; the *M*-, *I*-, and *D*-value computed in PE N in each iteration step are output. In the next pass the following N characters of the query sequence are loaded into the array. The data stored previously is loaded together with the corresponding subject sequences and sent again through the processor array. The process is iterated until the end of the query model is reached.

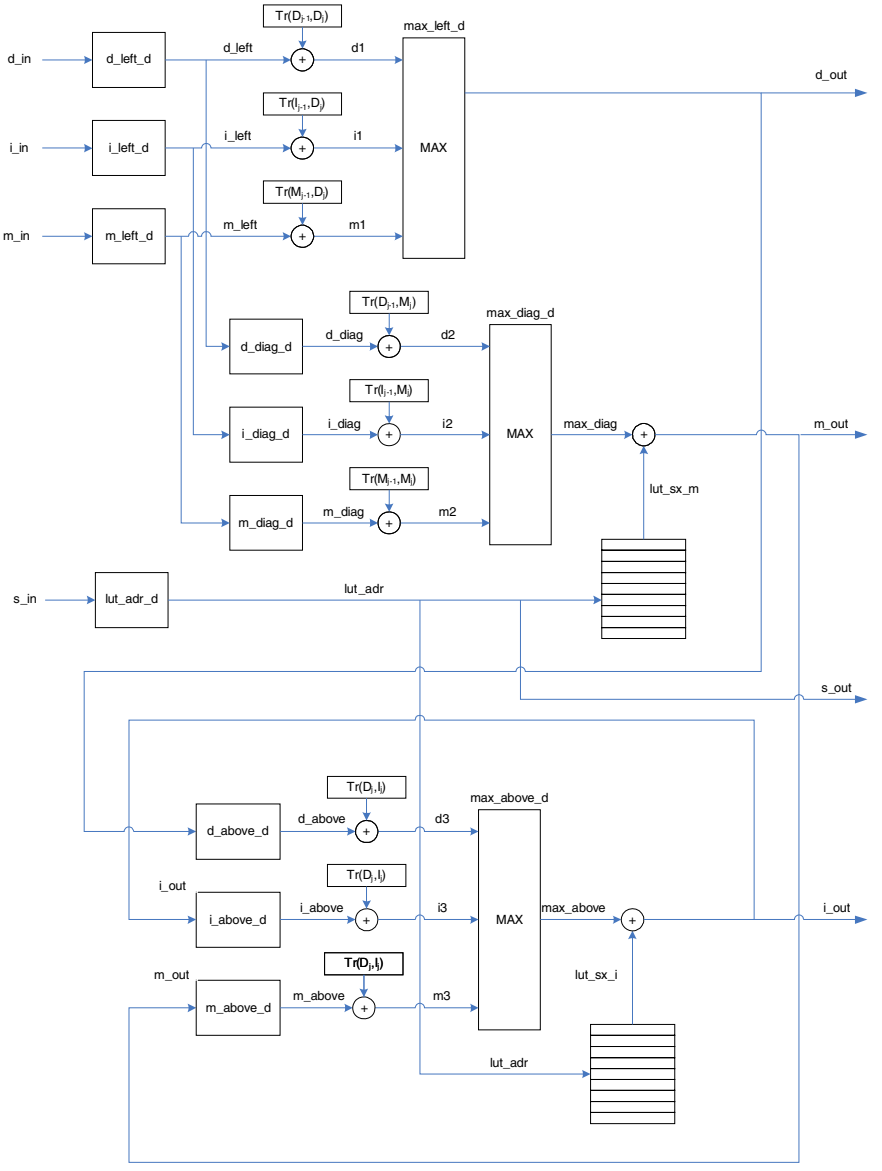


Fig. 6. Schematic diagram of our PE design

Unfortunately, this solution requires a large amount of memory (assuming 24-bit accuracy for intermediate results, nine times the database size per pass is needed). The memory requirement can be reduced by factor q by splitting the database into q equal-sized pieces and computing the alignment scores of all subject sequences within each piece. However, this approach also increases the loading time of substitution table

columns by a factor of q. In order to eliminate this loading time we have slightly extended our PE design. Each PE now stores emission tables and transitions probabilities for p HMM sates instead of only one. Although this increases the area per PE slightly, it allows for alignment of each database sequence with the complete query model without additional delays. It also reduces the required memory for storing intermediate results to nine the times longest database sequence size (again assuming 24-bit accuracy). Figure 7 illustrates this solution for 4 PEs.

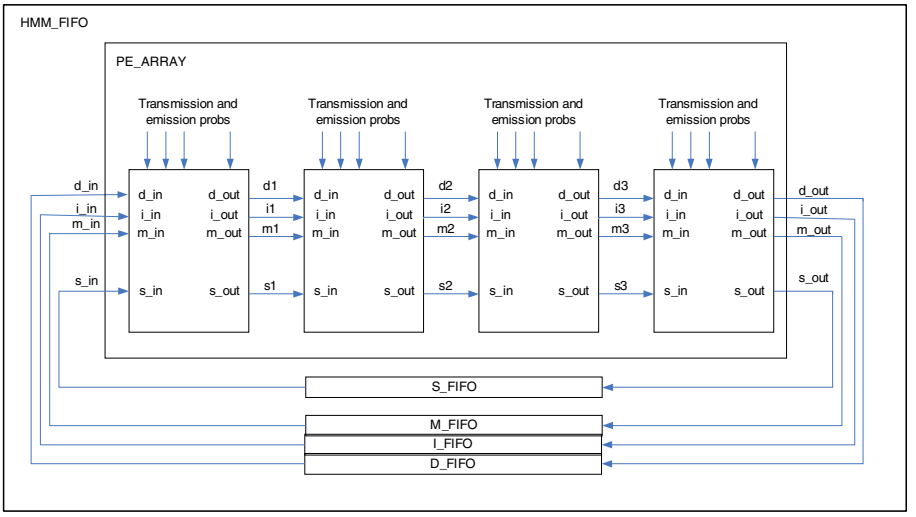


Fig. 7. System implementation

The database sequences are passed in from the host one by one through a FIFO. The database sequences have been pre-converted to LUT addresses. For query lengths longer than the PE array, the intermediate results are stored in a FIFO. The FIFO depth is sized to hold the longest sequence in the database. The database sequence is also stored in the FIFO. On each consecutive pass an LUT offset is added to address the emission table corresponding to the state of the next iteration step within the PEs.

5 Performance Evaluation

We have described the PE design in Verilog and targeted it to the Xilinx Virtex II architecture. We have specified an area constraint for each PE. The linear array is placed in a zigzag pattern as shown in Figure 8. We use on-chip RAM for the partial result FIFO, i.e. one column of block SelectRAM. The host interface also takes up some of the FPGA space in the bottom right-hand corner. Our design has been synthesized with Synplify Pro 7.0. We have used Xilinx ISE 6.3i for mapping, placement and routing.

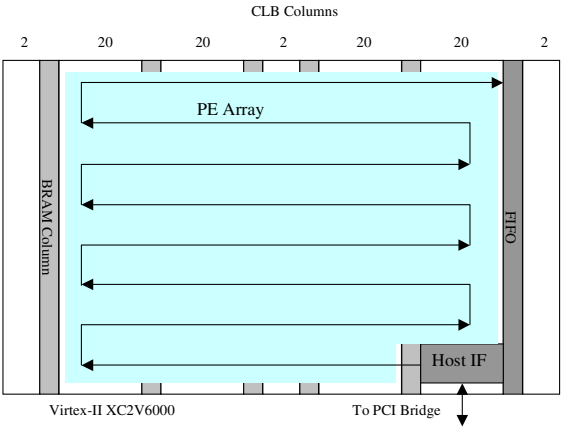


Fig. 8. System Floor plan in the XC2V6000 on the Alpha-Data ADM-XRC-II Board

The size of one PE is 30×8 CLBs. We have implemented a linear array of these PEs. Using a Virtex II XC2V6000 on an Alpha-Data ADM-XRC-II PCI board we are able to accommodate 36 PEs. The corresponding clock frequency is 37MHz.

A performance measure commonly used in computational biology is *cell updates per second* (CUPS). A CUPS represents the time for a complete computation of one entry of each of the matrices *M*, *D*, and *I*. The CUPS performance of our implementations can be measured by multiplying number of PEs times the clock frequency: 37 MHz × 36 PEs = 1.33 GCUPS. We have implemented the same application in the C programming language. Using MS Visual C++ 6.0 with optimizing compiler option (/Ox), it achieves a performance of 22 MCUPS on a Pentium IV 3GHz running Windows XP. Hence, our FPGA implementation has a speedup of approximately 60.

For the comparison of different massively parallel machines, we have taken data from [21] and [5]. Kestrel [5] and Systola [21] are one-board SIMD solutions. Our Virtex II XC2V6000 design is more than ten times faster the reported Viterbi implementations on Kestrel and more than 30 times faster than the one reported for Systola. These boards reach a lower performance, because they have been built with older CMOS technology (Kestrel: 0.5-µm, Systola 1024: 1.0-µm) than the Virtex II XC2V6000 (0.15-µm). Extrapolating to this technology both SIMD and reconfigurable FPGA platforms have approximately equal performance. However, the difference between both approaches is that FPGAs allow easy upgrading, e.g. targeting our design to a Virtex II XC2V8000 would improve the performance by around 30%.

6 Conclusions and Future Work

In this paper we have demonstrated that re-configurable hardware platforms provide a cost-effective solution to high performance biological sequence analysis with profile HMMs. We have described a partitioning strategy to implement database scans with the Viterbi algorithm on a fixed-size processor array and varying query model lengths. Using our PE design and our partitioning strategy we can achieve high

performance at low cost on an off-the-shelf FPGA. Our FPGA implementation achieves a speedup of approximately 60 as compared to a standard desktop computing platform.

The rapid growth of genomic databases demands even more powerful parallel solutions in the future. Because comparison and alignment algorithms that are favored by biologists are not fixed, programmable parallel solutions are required to speed up these tasks. As an alternative to inflexible special-purpose systems, hard-to-upgrade SIMD systems, and expensive supercomputers, we advocate the use of reconfigurable hardware platforms based on FPGAs.

Our future work includes extending our design to compute local alignment between a sequence and a profile HMM and making our implementation available to biologists as a special resource in a computational grid. We will be making the design more flexible at run-time. This requires the processors to be described using a language like Xilinx's RTPCore [10] specification which, in turn, uses the JBits API [11].

References

1. Altschul, S.F., Gish, W., Miller, W., Myers, E.W, Lipman, D.J.: Basic local alignment search tool, *J. Mol. Biol.*, 215 (1990) 403-410.
2. Bateman, A., et al: The PFAM Protein Families Database, *Nucleic Acid Research*, 32: 138-141, 2004.
3. Borah, M., Bajwa, R.S., Hannenhalli, S., Irwin, M.J.: A SIMD solution to the sequence comparison problem on the MGAP, *in Proc. ASAP'94*, IEEE CS (1994) 144-160.
4. Chow, E., Hunkapiller, T., Peterson, J., Waterman, M.S.: Biological Information Signal Processor, *Proc. ASAP'91*, IEEE CS (1991) 144-160.
5. Di Blas, A. et al: The UCSC Kestrel Parallel Processor, *IEEE Transactions on Parallel and Distributed Systems*, 16 (1) (2005) 80-92.
6. Durbin, R., Eddy, S., Krogh, A., Mitchison, G.: Biologcial Sequence Analysis, Probabilistic models of proteins and nucleic acids, *Cambridge University Press* (1998).
7. Eddy, S.R.: Profile Hidden Markov Models, *Bioinformatics*, 14 (1998) 755-763
8. Glemet, E., Codani, J.J.: LASSAP, a Large Scale Sequence compArison Package, *CABIOS* 13 (2) (1997) 145-150.
9. Gokhale, M. et al.: Processing in memory: The Terasys massively parallel PIM array, *Computer* 28 (4) (1995) 23-31.
10. Guccione, S.A., Levi, D.: Run-Time Parameterizable Cores, *Proc.* FPL'99, Springer, LNCS 1673, (1999) 215-222.
11. Guccione, S.A., Levi, D., Sundararajan, P.: JBits: A Javabased Interface for Reconfigurable Computing, *2nd Annual Military and Aerospace Applications of Programmable Devices and Technologies Conference (MAPLD Con'99)*.
12. Guerdoux-Jamet, P., Lavenier, D.: SAMBA: hardware accelerator for biological sequence comparison, *CABIOS* 12 (6) (1997) 609-615.
13. Hoang, D.T.: Searching genetic databases on Splash 2, in *Proc. IEEE Workshop on FPGAs for Custom Computing Machines*, IEEE CS, (1993) 185-191.
14. Hughey, R.: Parallel Hardware for Sequence Comparison and Alignment, *CABIOS* 12 (6) (1996) 473-479.
15. Lavenier, D., Pacherie, J.-L.: Parallel Processing for Scanning Genomic Data-Bases, *Proc. PARCO'97*, Elsevier (1998) 81-88.

16. Lopresti, D.P.: P-NAC: A systolic array for comparing nucleic acid sequences, *Computer* 20 (7) (1987) 98-99.
17. Muriki, K., Underwood, K.D., Sass, R.: RC-BLAST: Towards a Portable, Cost-Effective Open Source Hardware Implementation, *Proc. IPDPS'05*, IEEE Press, Denver, CO (2005)
18. Oliver, T., Schmidt, B., Nathan, D., Clemens, R., Maskell, D.: Multiple Sequence Alignment on an FPGA, Proc. ICPADS 2005 (Workshops), IEEE, Fukuoka, Japan, 2005.
19. Oliver, T., Schmidt, B., Maskell D.: Hyper Customized Processors for Bio-Sequence Database Scanning on FPGAs, Proc. FPGA'05, ACM, Monterrey, CA, 2005
20. Schmidt, B., Schröder, H., Schimmler, M: Massively Parallel Solutions for Molecular Sequence Analysis, *Proc. 1st IEEE Int. Workshop on High Performance Computational Biology,* Ft. Lauderdale, Florida, 2002
21. Schmidt, B., Schröder, H.: Massively Parallel Sequence Analysis with Hidden Markov Models, *International Conference on Scientific & Engineering Computation* (IC-SEC 2002), World Scientific, Singapore, 2002
22. Singh, R.K. et al.: BIOSCAN: a network sharable computational resource for searching biosequence databases, CABIOS, 12 (3) (1996) 191-196.
23. Smith, T.F., Waterman, M.S.: Identification of common molecular subsequences*, J. Mol. Biol.* 147 (1981) 195-197.
24. TimeLogic Corporation, http://www.timelogic.com.
25. Yamaguchi, Y., Maruyama, T., Konagaya, A.: High Speed Homology Search with FPGAs, *Proc. Pacific Symposium on Biocomputing'02*, pp.271-282, (2002).
26. Yu, C.W., Kwong, K.H., Lee, K.H., Leong, P.H.W.: A Smith-Waterman Systolic Cell, *Proc. FPL'03*, Springer, LNCS 2778, (2003) 375-384.

FPGAs for Improved Energy Efficiency in Processor Based Systems

P.C. Kwan and C.T. Clarke

Department of Electronic and Electrical Engineering,
The University of Bath, Claverton Down, Bath, UK

Abstract. Processor based embedded systems often need to apply simple filter functions to input data. In this paper we examine the relative energy efficiency of microprocessor and Field Programmable Gate Array (FPGA) implementations of these functions. We show that considerable savings can be achieved by adding an FPGA to a microprocessor based systems and propose a strategy to reduce the impact of the excessive leakage energy overhead in low data rate applications.

1 Introduction

Processor based embedded systems are used to collect data in a wide range of applications. The first stage of processing this data is often the application of finite or infinite impulse response filters to remove noise and constrain the frequency range of the input [1]. Where the input data rate or the filter order is high, the use of reconfigurable logic such as Field Programmable Gate Arrays (FPGAs) and Complex Programmable Logic Devices (CPLDs) within processor based embedded systems is often justified on the basis of performance. There is a clear case for inclusion of such technologies in order to meet system design requirements where input data rates are higher than the processor can cope with on its own.

There are, however, many applications in which speed of operation is within reach of a microprocessor and reducing power consumption is the dominant design aim. In these applications, FPGAs and CPLDs are usually excluded from the system design as it is assumed that adding these components will increase power consumption. It is clear that the inclusion of one of these devices can allow the processor clock frequency to be reduced and we have investigated the trade-offs of including a programmable logic device within the system when power reduction for a fixed required performance level is the designers aim.

In this paper we show that the power savings achieved more than offset the power consumed by the gate array. We also examine the impact of the variation in the ratio of static to dynamic power consumption caused by increased leakage currents in deep sub-micron processes [2]. The results of this investigation are used to propose a strategy for maximising energy efficiency of filtering operations in low rate data systems making use of switched power supplies for memories and logic arrays. We show that increased configuration energy requirements in power switched FPGAs can be less than the reduction in leakage energy achieved by that power switching.

T. Srikanthan et al. (Eds.): ACSAC 2005, LNCS 3740, pp. 440 – 449, 2005.

2 Basis of Comparison

In order to compare processor and FPGA based filtering operations it is necessary to set clear criteria for the filtering operations. In this paper we consider a range of different filters sizes but all filters considered have both 16 bit data and coefficients. All filters are constructed using a number of identical second order stages. Each stage is generated on the FPGA using a fully parallel single cycle filter stage implementation. On the processor, a C implementation provided an almost identical design complexity making the two designs comparable in terms of the filter quality vs. designer effort. Table 1 compares the core filter code for the C and VHDL implementations. For the sake of clarity, sign extension and bit selection has been omitted from the VHDL code sample.

The tools used to convert the FIR filter code into an implementation were the freely available design tools published by the silicon vendor. In the case of the micro-controller this was the NEC V800 series evaluation kit compiler and in the case of the FPGA the tools used was Xilinx ISE.

The leakage energy is a significant issue as the devices considered range over a number of process technologies. For this reason, the current flow into both FPGAs and processors is measured when the clock speed for the device has been set to give exactly the sample rate required. This leads to a higher leakage overhead being associated with filtering operations at a low rate as leakage is not related to the clock frequency.

Table 1. A comparison of the core FIR filter code used on the processor and FPGA

C	VHDL

`d_x[2] = d_x[1];`
`d_x[1] = d_x[0];`
`d_x[0] = x;`
 	
`for(i=0 ; i<3 ; i++)`
`  y += b[i]*d_x[i];`
	`if rising_edge(clk) then`
	`  x (2) <= x (1);`
	`  x (1) <= x (0);`
	`  x (0) <= x_in;`
	`  for i in 0 to 2 loop`
	`  p(i) := x(i)*b(i);`
	`  end loop;`
	`  y <= p(0)+p(1)+p(2);`
	`end if;`

When comparing the energy and power values, it is necessary to allow for all the components in a minimum system. In the processor case, the power consumption is measured including the necessary memory and interface circuits as this represents a minimum usable system. In the case of the FPGA, external memories are not required so the only the total FPGA power is considered.

For the purposes of comparison, energy is measured in Joules per sample per filter stage. This gives a measure that can be compared directly for different logic and processor devices and permits comparison of energy efficiency of filters at different sample rates, and with different numbers of stages.

3 Energy Usage in a Processor

As the basis of our comparison, we take a NEC V-853 microcontroller. This is a simple microcontroller with bit level I/O capabilities and this made interfacing simple. This processor does not have a hardware multiplier. We used the standard compiler supplied be NEC, and programmed the device in C. This gave us an approximate equivalence with the design complexity and effort applied in the gate array designs. It is clear that different processors would give different results but this processor is available for similar prices to the FPGAs considered and is suitable for use in low data rate applications that we are examining in this paper.

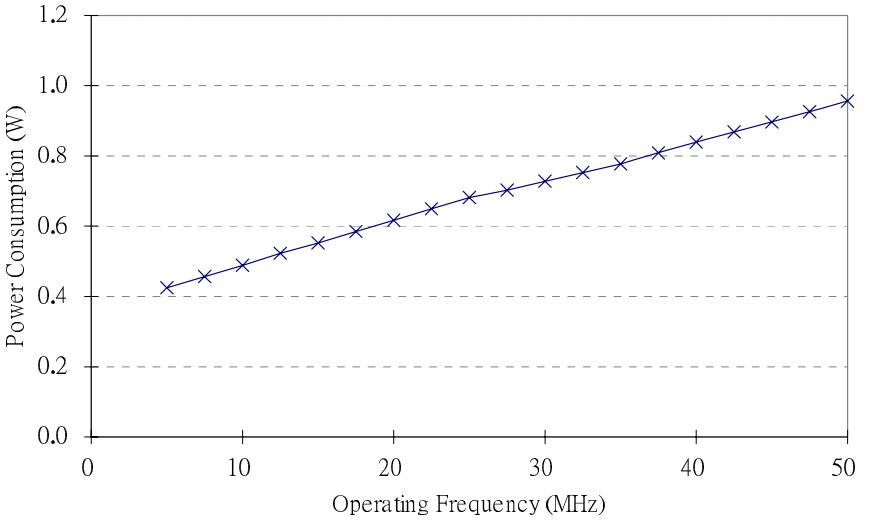


Fig. 1. Power consumption in the V-853 vs. clock frequency

Figure 1 shows the power consumption for the V-853 when running a single standard second order FIR filter stage. The power consumption is made up of a static component of about 350mW and a dynamic power consumption of around 12mW/MHz. The FIR filter loop takes 3035 cycles to run. This corresponds to an asymptotic filtering energy of 36μJ/Sample/Stage. At low frequencies (below about 27MHz) leakage and other static power consumers dominate the power consumption.

4 Energy Usage in FPGAs

Three Xilinx FPGAs are used to create a comparison with the processor results given above. Figure 2. shows the power consumption of the different FPGAs with

frequency. Of the FPGAs compared, the Virtex and Spartan 2 do not have a hardware multiplier block whereas the Virtex 2 does. The Virtex device is the best match to the V-853 in terms of fabrication technology [3,4] and thus provides the best basis for comparison. The power consumption is plotted for 2 stages and for the maximum number of stages that can be placed in the FPGA used. In the case of the Virtex device (an XCV50) only 2 stages can be placed on the FPGA. Examining the change in power consumption of the Spartan 2 (an XS2V100) and the Virtex 2 (an XC2V250) it can be seen that increasing the number of filter stages does not lead to a linear increase in power dissipation. This is due to the power consumption of idle circuitry in the FPGA when a small number of stages is used. For example, in the case of the Virtex 2 device implementing 2 filter stages, less than 25% of the device is being used. When the device is filled, there are 8 stages, but the power is only (roughly) doubled.

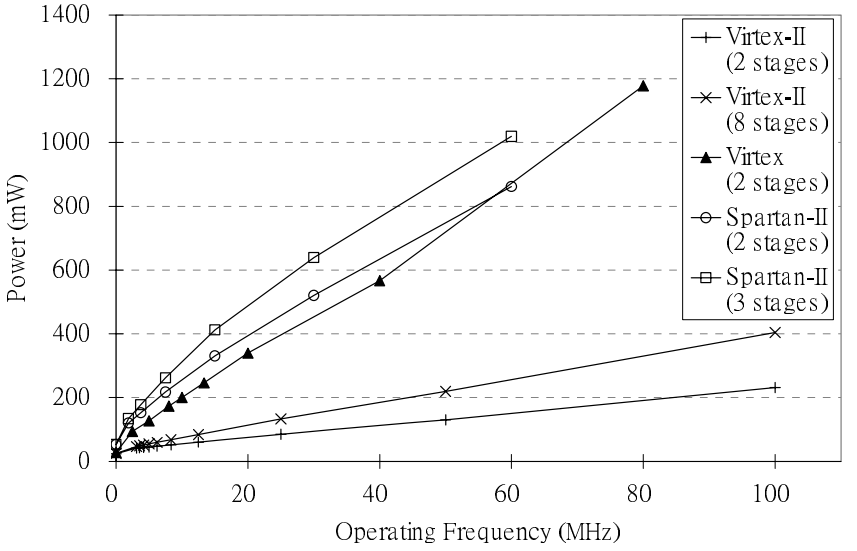


Fig. 2. Power consumption in a range of FPGAs vs. clock frequency

The significance of this observation in terms of low power operations is that an FPGA that has spare capacity will waste a significant amount of energy and it is therefore very important to use an FPGA that is only just big enough for the application. It is possible to plot the normalised power consumption against the utilisation of the FPGA as shown on figure 3. This graph shows the average value and 1 standard deviation error bars. Whilst there is a significant variation, it can be seen that around one quarter of the dynamic power consumption is independent of utilisation. This value is consistent with the level of power consumption that might be expected in structures such as a global clock. As FPGA sizes increase in quite large steps (at least at the low end of the product range). The implication is that if a design just fits in a an XCV50, then the power consumption penalty of using an XCV100 instead (a common approach to give design headroom in case of changes) would be around 35%.

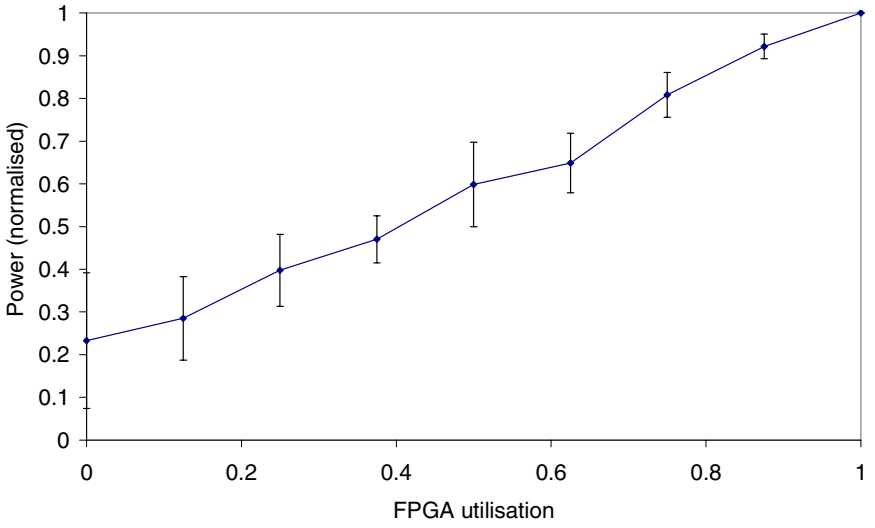


Fig. 3. The impact of utilisation on Normalised power consumption

The Virtex device consumes approximately a static power of 28mW and a dynamic power of around 14.5mW/MHz. Static energy losses therefore dominate below 2MHz. However these values are not directly comparable to the results of the processor since the FPGA will process one sample per cycle, and the FPGA is executing 2 stages. Thus the asymptotic energy consumption is 7.25nJ/Sample/Stage. Given the utilisation estimations above the dynamic power consumption for a single stage on the same FPGA would be around 9mW/MHz which has asymptotic energy consumption of 9nJ/Sample/Stage.

5 FPGAs Within Processor Based Systems

FPGAs are becoming popular as embedded components in computing platforms, but are generally considered to have poor energy efficiency. However, when we consider the results shown in this paper, it is clear that this is not necessary true when comparing with microprocessors.

Figure 4 compares the energy usage of the FPGA and microcontroller implementations of the same filter. Plotted against sample rate rather than clock rate makes the comparison fair as the FPGA and processor are being required to do the same function at the same rate. It can be seen that the FPGA is far more energy efficient than the processor at all data rates. It achieves this energy efficiency as a direct result of being able to provide the same filtering function with a very much lower clock rate. All the points marked on the graph are the results of direct measurements of current flow to ground from the microcontroller/FPGA. A variable frequency clock source was used to set the system to just achieve the required sample rate.

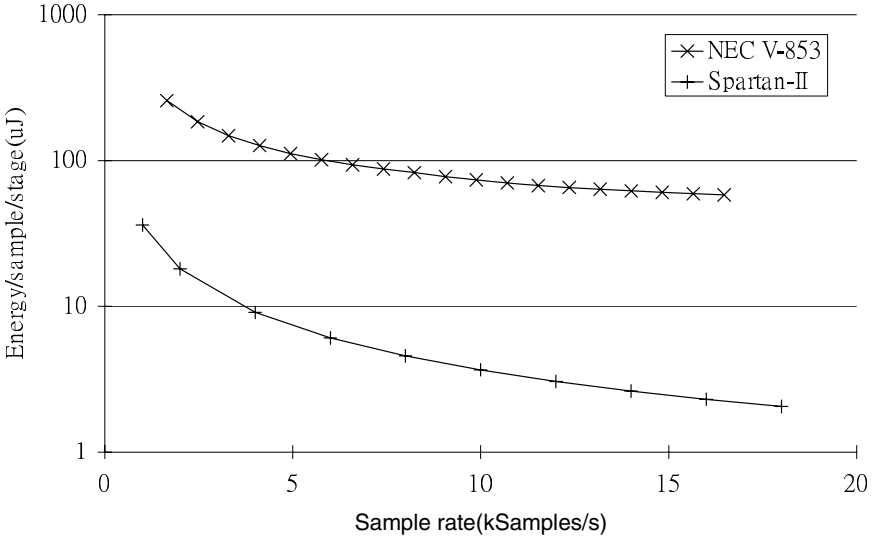


Fig. 4. A comparison of FPGA and microcontroller energy usage for FIR filtering

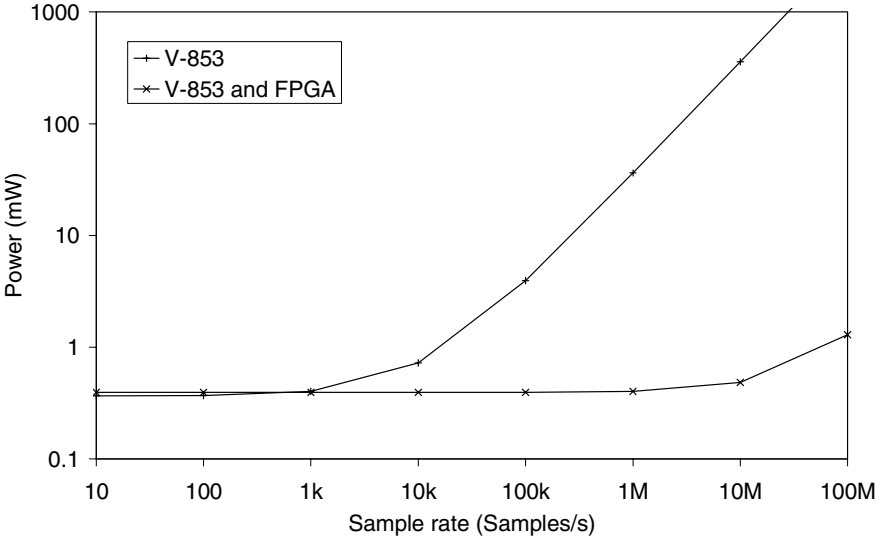


Fig. 5. Power consumption with and without an FPGA

As was previously mentioned, the increase in energy usage per sample per stage at low sample rates is due to the leakage energy which dominates at these rates. Given the energy saving available it is possible to consider placing an FPGA in a low data rate circuit to provide a filtering function that offloads the processor thereby reduces power consumption of the system as a whole. If we extrapolate from the results above, it is possible to compare the power consumption of a processor based system

with and without an FPGA to perform a single filtering operation. When an FPGA is used with the FPGA is used in conjunction with the processor, it is expected that the FPGA would operate as a co-processor, possibly with direct access to a memory device or the data source (such as an ADC).

Figure 5 shows the system power consumption for each of these situations. In the figure, it is assumed that there is no other load on the microcontroller, but if there were, the impact would simply be to shift the whole graph upwards by the dynamic power required for that function. For the purposes of extrapolation, it has been assumed that the FPGA (a Xilinx Spartan-II xc2s100) is only 50% utilized as this is a sensible worst case given the typical available variants in FPGA families. In addition, the power consumption for the microcontroller on it's own has been extrapolated to data rates higher than it is actually capable of. This is simply to show the difference between the two approaches.

From figure 5 it is clear that power savings are available down to data rates of approximately 1kSamples/s. The power saving at 10kSamples/s is about 45%. This lower limit for power improvements is due to the leakage power of the FPGA and so, if the FPGA is on all the time, this is a hard limit on the lowest data rate at which energy savings can be achieved.

5.1 Power Switching

It has been shown that an FPGA can be more energy efficient than a microcontroller at any given medium or high data rate (above 1kSamples/s in our specific example). However, it should also be noted that at low data rates, the energy per sample per stage is dominated by leakage. In the case of the Xilinx Spartan-II xc2s100 at 1kSamples/s, more than 99.9% of the energy consumed is accounted for by the leakage. The FPGA is more energy efficient when running at high frequencies as less leakage energy would be wasted per sample. By making use of this characteristic, further efficiencies can be realised.

Leakage energy can only be reduced by processing the data in batches, and switching off the FPGA in between these batches. However, this has three significant implications:

1. Data captured whilst the FPGA is switched off must be stored somewhere until the FPGA is available to process it.
2. The filtering process will incur additional latency directly proportional to the batch size. Whether this is an issue of not depends entirely on the application.
3. Each time the FPGA is switched on, it must be configured, and reloaded with its state.

There are therefore two power consumers that need to be considered when examining the possibility of switching the power to the FPGA; data storage memory and FPGA configuration.

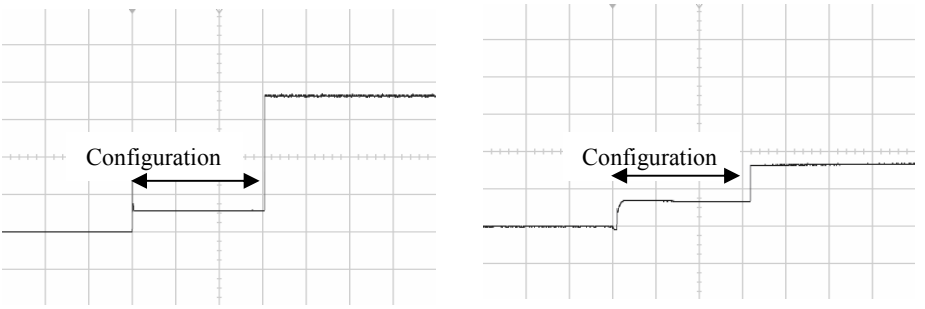
5.2 Data Storage Memory

Data sampled at a low rate may be stored in a general purpose memory. However, low power memory cell have been proposed [5,6,7] that can be placed into a "drowsy" or

"sleep" states in which data is held but is not accessible. Given the very regular memory access patterns, it is possible to predict which SRAM cells will be accessed and power consumption of the SRAM can thus be optimised. Using a conventional SRAM we would expect to be able to store 1000 entry blocks of data for a 1kSample/s data input and access it using an optimally sized SRAM consuming about 0.5mW. However, using the regular nature of the accesses would allow a custom memory to operate at around 1% of this value (i.e. 5µW). This is because 99% of the memory would be in sleep mode at any given time and the power taken by these cells would be a factor 10^{-3} times less than a conventional SRAM [7].

5.3 FPGA Configuration

When an FPGA is powered-up, it needs to be configured before it can operate. The configuration process is a serial read from a dedicated FLASH memory. Only once this is completed can the FPGA load the previous FIR filter state and process the next data block. The energy required to configure an FPGA is not a readily available value, and will obviously vary considerably from one FPGA to another. Once it is configured, the FPGA can almost immediately process data at high speed. In order to evaluate the configuration energy of the FPGA, 0.3Ω resistors were used in series with the power supplies to the FPGA and FLASH configuration memory and the device started. Figure 6 shows the oscilloscope traces for the internal power supply (Vccint) and external power supply (Vcco) current measuring resistors. Vcco (3.3V) also supplies the Flash memory on the board used.



a)Iccint (67mA/div vertically) b) Icco (33mA/div vertically)

Fig. 6. Configuration current of a Spartan II xc2s100 FPGA (50ms/div horizontally)

The configuration energy for this particular Spartan-II is approximately 25mJ. This is equivalent to the leakage energy over a period of 0.5s. This configuration used a simple serial load from a Xilinx Flash memory using the clock source generated automatically by the FPGA. It is possible to use an external clock source to configure the FPGA faster, but this was not done as it would potentially add an additional clock source to the design which would itself consume power.

5.4 Energy Savings Due to Block Based FPGA Filtering

Due to the low power consumption of the SRAMs described above, it is possible to consider a scheme in which data is stored in an SRAM until a sufficiently large block is available, and then the FPGA is switched on and configured, the filter runs, and the FPGA is switched off again. For low data rate input signals for which a latency of more than about 0.5s is acceptable, this will generally reduce total power consumed.

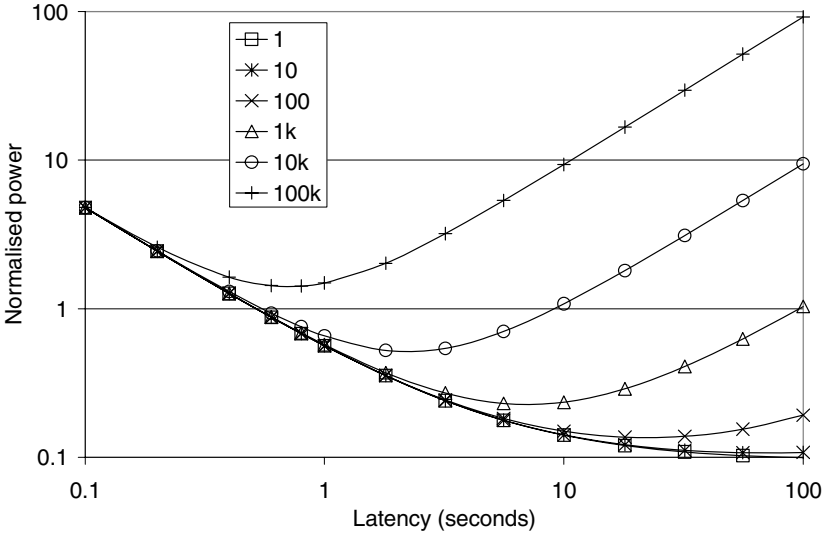


Fig. 7. Normalised power consumption of an FPGA and low power SRAM for block based operation for a range of data input rates

Figure 7 shows the power consumption as a function of latency due to the block based approach. Lines are drawn for a range of input data rates from 1 to 100,000 samples/s and the power is normalised against the power consumed in a continuously operating FPGA at the data rate for a given line.

For low latencies (less than 0.5 seconds), the reconfiguration energy required is more than the leakage energy saved. At high latencies (input data rate dependant) the power required to maintain a large SRAM with all the data samples awaiting processing is greater than the leakage power in the FPGA. Between these extremes, there is a "sweet spot" at which power efficiency is optimised although this optimal point may not actually represent a power saving as is the case for 100kSamples/s in figure 7. As the data rate goes down, the efficiencies possible increase, as does the optimal latency.

6 Conclusions

In this paper we have discussed the power savings possible by using an FPGA in a processor based system with inputs that required filtering. Even at very low data rates,

FPGAs offer savings, and if latency is acceptable in the application, the FPGA could be power switched and combined with a low power SRAM to further reduce power consumption for the low rate data input filtering.

Our results are based upon measurements made on existing production devices and the actual savings possible will depend on the actual device used and issues such as the overall utilization of the FPGA. However these differences are likely to be small in comparison to the power savings possible by adding an FPGA.

References

1. Tian-Sheuan Chang; Yuan-Hua Chu; Chein-Wei Jen, "Low-power FIR filter realization with differential coefficients and inputs", IEEE Transactions on Circuits and Systems II, Volume 47, Issue 2, Feb. 2000 Page(s):137 – 145.
2. The International Technology Roadmap for Semiconductors, http://public.itrs.net/
3. Wim Roelandts "15 Years of Innovation", Xilinx Xcell Issue 32 Quarterly Journal (online: http://www.xilinx.com/xcell/xl32/xl32_4.pdf).
4. "Release of World's First 32-bit RISC Microcontroller Having Flash Memory", NEC News Release (online: http://www.necel.com/english/news/9603/1202.html).
5. Flautner, K., Nam Sung Kim, Martin, S., Blaauw, D., Mudge, T., "Drowsy caches: simple techniques for reducing leakage power", 29th Annual International Symposium on Computer Architecture, Proceedings, 25-29 May 2002 Page(s):148 – 15.
6. Nam Sung Kim, Flautner, K., Blaauw, D., Mudge, T., "Drowsy instruction caches. Leakage power reduction using dynamic voltage scaling and cache sub-bank prediction", International Symposium on Microarchitecture (MICRO-35), Proceedings, 18-22 Nov. 2002 Page(s):219 – 230.
7. Nii, K., Makino, H., Tujihashi, Y., Morishima, C., Hayakawa, Y., Nunogami, H., Arakawa, T., Hamano, H., "A low power SRAM using auto-backgate-controlled MT-CMOS", International Symposium on Low Power Electronics and Design, 1998. Proceedings. 10-12 Aug 1998, Page(s):293 – 298.

Morphable Structures for Reconfigurable Instruction Set Processors

Siew-Kei Lam, Deng Yun, and Thambipillai Srikanthan

Centre for High Performance Embedded Systems,
Nanyang Technological University,
Nanyang Drive, Singapore 637553
{assklam, pg10831880, astsrikan}@ntu.edu.sg

Abstract. This paper presents a novel methodology for instruction set customization of RISPs (Reconfigurable Instruction Set Processors) using morphable structures. A morphable structure consists of a group of hardware operators chained together to implement a restricted set of custom instructions. These structures are implemented on the reconfigurable fabric, and the operators are enabled/disabled on demand. The utilization of a predefined set of morphable structures for instruction set customization dispenses the need for hardware synthesis in design exploration, and avoids run-time reconfiguration while minimizing the reconfigurable area. We will describe the two stages of the methodology for constructing the morphable structures, namely template generation and identification of a maximal unique pattern set from the templates. Our preliminary studies show that 23 predefined morphable structures can sufficiently cater to any application in a set of eight MiBench benchmark applications. In addition, to achieve near-optimal performance, the maximum required number of morphable structures for an application is only 8.

1 Introduction

Future embedded systems will require a higher degree of customization to manage the growing complexity of the applications. At the same time, they must continue to facilitate a high degree of programmability to meet the shrinking TTM (Time To Market) window. Lately, extensible processors [1], [2] have emerged to provide a good tradeoff between efficiency and flexibility. Many commercial processors (e.g. Xtensa from Tensilica [3], ARCtangent from ARC [4], etc.) offer the possibility of extending their instruction set for a specific application by introducing custom functional units within the processor architecture. This application-specific instruction set extension to the computational capabilities of a processor, provides an efficient mechanism to meet the growing performance and TTM demands of embedded systems. However, the NRE (Non-Recurring Engineering) costs of redesigning a new extensible processor can still be quite high. This is exacerbated as the cost, associated with design, verification, manufacture and test of deep sub-micron chips, continue to increase dramatically with the mask cost.

A RISP (Reconfigurable Instruction Set Processor) [5] consists of a microprocessor core that has been extended with a reconfigurable fabric. Similar to

T. Srikanthan et al. (Eds.): ACSAC 2005, LNCS 3740, pp. 450 – 463, 2005.

extensible processors, the RISP facilitate critical parts of the application to be implemented using a specialized instruction set on reconfigurable functional units. The advantages of a RISP over the extensible processors stem from the reusability of its hardware resources in various applications without incurring high NRE costs. Due to this, RISP are more flexible than an extensible processor, which precludes post design flexibility. However, reconfigurability of the RISP incurs an overhead that can hamper its ability to outperform conventional instruction set processors.

In this paper, we introduce a methodology for instruction set customization on RISPs that relies on a set of morphable structures to implement the custom instructions. A one-time effort is required to identify a unique set of morphable structures from a subset of enumerated custom instruction instances. The pattern enumeration method introduced in [6] is combined with graph isomorphism [7] to identify unique custom instruction instances from a set of embedded applications. The process of selecting a subset of custom instruction instances or templates is called template generation. We present a heuristic approach for selecting the templates from the enumerated patterns, and show that only a limited number of templates are required to achieve comparable results with known techniques.

The morphable structures are then constructed by using a subgraph isomorphism method to combine the selected templates into a set of maximal unique structures. These morphable structures are then characterized to obtain their hardware performance and cost models to be used for future design exploration. We show that the total number of unique morphable structures generated from eight applications in the MiBench embedded benchmark suite [8], is only 23. Moreover, a maximum of only 8 morphable structures are required for a particular application. This restricted set of morphable structures is sufficient to achieve high performance gain, while keeping the reconfigurable logic area low.

The availability of a predefine set of morphable structures dispenses the need for hardware implementations during design exploration. This can significantly increase the efficiency of the custom instruction selection process. The main goal of custom instruction selection is to determine viable custom instruction candidates from the application DFG (Data Flow Graph) to be implemented on a morphable structure. The custom instruction selection process in the methodology employs template matching that utilizes the restricted set of templates for improved efficiency. In this paper, we will not discuss the custom instruction selection process and limit the scope to the construction of morphable structures.

In the following section, we discuss some previous work in the areas of RISP and instruction set customization. In Section 3, we describe the notion of morphable structures on RISP and in Section 4, present our methodology for instruction set customization using these structures. Section 5 presents the experimental results, and the paper concludes with some consideration on future directions.

2 Background

An inherent problem in RISP arises from the reconfiguration overhead that is incurred while reusing the hardware resources for various functions. For example, the DISC (Dynamic Instruction Set Computer) processor proposed in [9] requires a

reconfiguration time that is projected to contribute up to 16% of an application's total execution time. In [10], a compiler tool chain was presented to encode multiple custom instructions in a single configuration to reduce the reconfiguration overhead and maximize the utilization of the resources. However, the compiler tool chain incorporates a hardware synthesis flow that hampers the efficiency of the design exploration process.

In commercial RISPs, the run-time reconfiguration overhead is exacerbated by the fine-grained programmable structure. For example, the Stretch processor [11] requires 80μs to change an instruction on their proprietary programmable logic. In order to maximize the efficiency of hardware execution, commercial RISPs [12], [13] often provide a large reconfigurable area to accommodate all the custom instructions of an application. These custom instructions are implemented on the reconfigurable fabric prior to execution to avoid run-time reconfiguration. However, these processors are likely to violate the tight area constraints imposed by most embedded systems.

For a given application, a RISP configuration that outperforms the conventional processors must be determined rapidly without delaying the short TTM requirements for embedded systems. However, automatically determining the right set of extensible instructions for a given application and its constraints remains an open issue [2]. The problem of custom instruction identification can be loosely described as a process of detecting a cluster of operations or sub-graphs from the application DFG to be collapsed into a single custom instruction to maximize some metric (typically performance). Previous works in custom instruction identification can be broadly classified into the following four categories: 1) pattern matching [14], 2) cluster growing [15], 3) heuristic-based [16], and 4) pattern enumeration [6].

In [14], an approach that combines template matching and generation have been proposed to identify clusters of operations based on recurring patterns. The clusters identified with this approach are typically small and may not lead to a notable gain when implemented as custom instructions. The method proposed in [15] attempts to grow a candidate sub-graph from a seed node. The direction of growth relies on a guide function that reflects the merit of each growth direction. In [16], a genetic algorithm was devised to exploit opportunities of introducing memory elements during custom instruction identification.

The methods discussed above have demonstrated possible gains, but they can potentially miss out on identifying some good custom instruction candidates. The pattern enumeration method proposed in [6] employs a binary tree search approach to identify all possible custom instruction candidates in a DFG. In order to speed up the search process, unexplored sub-graphs are pruned from the search space if they violate a certain set of constraints (i.e. number of input-output ports, convexity, operation type, etc.). In [17], pattern enumeration is combined with pattern generation and matching to identify the most profitable custom instructions in an application. Although these two approaches can lead to promising results, they can still become too time-consuming especially when dealing with large applications.

The methodology proposed in this paper differs from previously reported work as it aims to identify a predefine set of morphable structures that can implement custom instructions of numerous applications. Unlike existing methods (i.e. [6], [17]), which employs a time-consuming pattern enumeration process for each application, the proposed technique performs this process only once on a standard set of applications. The enumerated patterns are used to generate a set of morphable structures, which are

then characterized to obtain their hardware properties. The pre-characterized structures lead to substantial reduction in the design time, as it does not necessitate a lengthy hardware synthesis process during application mapping such as that required in existing methods (i.e. [10]). Our preliminary studies show that only a small number of morphable structures can sufficiently cater to eight embedded applications, while providing comparable performance gain with existing techniques. In addition, this study opens up new possibilities for area-efficient designs of commercial RISPs [12], [13], as the proposed methodology provides an insight to the reconfigurable area needed for efficient custom instruction implementations.

3 Instruction Set Customization Using Morphable Structures

A morphable structure consists of a group of operators that are chained together to implement a restricted set of custom instructions. These operators are derived from the set of primitive operations in the processor's instruction set.

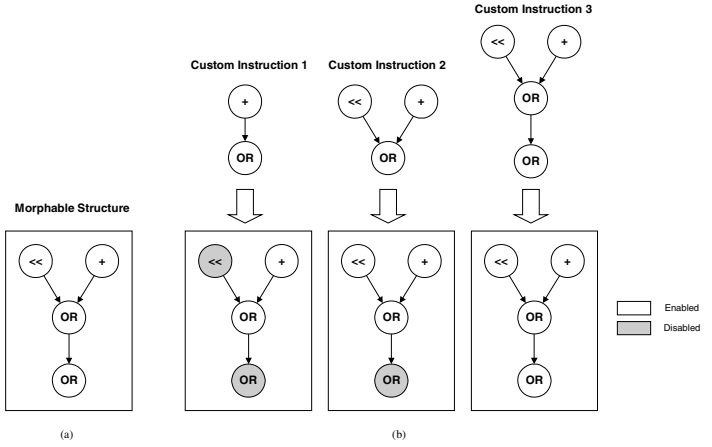


Fig. 1. Implementation of custom instructions on a morphable structure

Fig. 1a) illustrates an example of a morphable structure and Fig. 1b) describes how it can be used to implement three different custom instructions. Each of the custom instruction is sub-graph isomorphic to the structure. These custom instructions can be efficiently mapped onto the morphable structure by enabling and disabling the necessary operators. Operators that are disabled allow the input operand to bypass the primitive operation, and directly routed to the output port.

Fig. 2 shows an example of a RISP, which is four-wide VLIW (Very Long Instruction Word) architecture that has been extended with a reconfigurable fabric for implementing custom instructions. The morphable structures (denoted as MS) implemented on the reconfigurable fabric obtain their input data from the integer unit's register file, and outputs the results to an arbitrator. High-speed arbitrators such as that found in the Altera Nios configurable platform [13] are commonly used

to facilitate the sharing of register files or memory between the processor core and other peripherals. In the example RISP, the arbitrator is used to share the integer unit's register file between the ALU and custom logic. It is evident that the number and complexity of the morphable structures will affect the required area of the reconfigurable fabric. In addition, since the complexity of the arbitrator logic is dependent on the number of connections to the morphable structures, it is imperative to keep the number of morphable structures tractable.

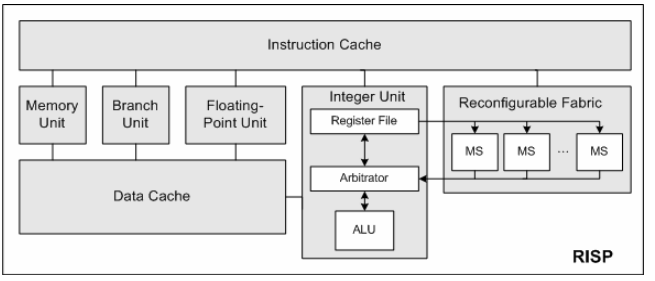


Fig. 2. Implementing morphable structures in a RISP

The morphable structures are pre-defined from a set of custom instruction instances obtained by enumerating a number of embedded applications. Since the morphable structures are derived from the custom instruction instances, they are likely to implement a large variation of custom instructions in embedded applications. This is a plausible assumption as it has been shown that domain-specific applications exhibit common dataflow sub-graph patterns [18], [19]. It is noteworthy that although a substantial amount of effort is required to obtain the morphable structures, this process is performed only once. In a later section, we will describe an approach to obtain the maximal unique set of morphable structures in a tractable manner.

The advantage of using morphable structures stems from the availability of a predefine set of morphable structures that can lead to rapid design exploration without a time-consuming hardware synthesis flow to evaluate the feasibility of the custom instruction candidates. This is possible as the morphable structures can be pre-characterized to facilitate area-time estimations of the custom instructions on hardware. In addition, a minimal set of morphable structures can be mapped onto the reconfigurable logic prior to the application execution to avoid run time reconfiguration. The reconfigurable logic space to accommodate the morphable structures is also minimized, as the number of morphable structures that are specific to a particular application is reasonably small.

4 Proposed Methodology

Fig. 3 illustrates an overview of the proposed methodology for instruction set customization using morphable structures.

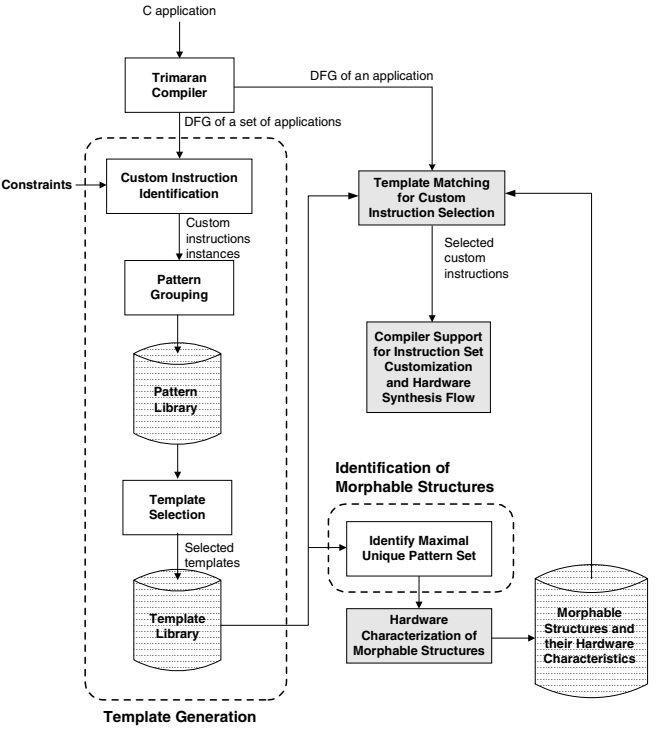


Fig. 3. Overview of proposed methodology

The proposed methodology consists of several key stages, but in this paper, we limit the discussion to the construction of morphable structures. This comprises of two stages: template generation and identification of morphable structures. It is noteworthy that these two stages along with the hardware characterization of morphable structures is a one-time process, whereas custom instruction selection performs template matching to select the custom instructions for each new application. The selected custom instructions and the corresponding morphable structures are passed to the compiler and hardware synthesis flow, which are beyond the scope of this paper. In the following sub-sections, we will provide more detailed descriptions of the two stages to construct morphable structures.

4.1 Template Generation

The main task of this stage is to perform template selection from a subset of custom instruction instances. The templates are used for two purposes. Firstly, the templates form the basic structures to construct the morphable structures, and secondly the templates are used to select custom instruction candidates from a given application.

It is noteworthy that compared to [17], the template generation process in our methodology is performed only once from a set of embedded applications. Let's

denote this set of embedded applications as the standard application set. Hence, although this process can be time-consuming due to pattern enumeration of large applications, it does not affect the custom instruction selection process.

The proposed approach is divided into three steps: 1) Custom instruction identification and 2) Pattern grouping, and 3) Template selection.

1) Custom Instruction Identification

The objective of this step is to enumerate the custom instruction instances from an application's DFG. We have modified the pattern enumeration algorithm in [6] to identify all the custom instruction instances from the standard application set. As mentioned earlier, the method in [6] employs a binary tree search approach that prunes unexplored sub-graphs from the search space if they violate a certain set of constraints.

We have used the Trimaran [20] IR (Intermediate Representation) for custom instruction identification. In order to avoid false dependencies within the DFG, the IR is generated prior to register allocation. For the purpose of this study, we have imposed the following constraints on the custom instructions to increase the efficiency of the identification process:

1. Only integer operations are allowed in the custom instruction instance.
2. Each custom instruction instance must be a connected sub-graph.
3. Maximum number of input ports 5 and maximum number of output ports 2. Previous work [21] has shown that input-output ports more than this range results in little performance gain when no memory and branch operations are allowed in the custom instructions.
4. Only convex sub-graphs [6] are allowed in the custom instructions instance.
5. The operation that feeds an input to the custom instruction instance must execute before the first operation in the custom instruction instance.

2) Pattern Grouping

The custom instruction instances are subjected to pattern grouping, whereby identical patterns that occur in different basic blocks and applications are grouped to create a unique set of custom instruction patterns. Patterns are considered identical if they have the same internal sub-graphs, without considering their input and output operands. The static occurrences of each unique pattern are also recorded. We have used the graph isomorphism method in the vflib graph-matching library [22] for the pattern grouping process. Due to the limited size of the constrained custom instruction instances, the pattern-grouping step can be accomplished rapidly.

These unique custom instruction patterns are stored in the pattern library for the template selection process. Fig. 4 presents the static occurrences and the corresponding pattern size of the unique patterns in the pattern library. The pattern size is calculated as the number of operations in the custom instruction. It can be observed that custom instructions with small pattern sizes occurs more frequently in the set of embedded applications as compared to custom instructions with large pattern sizes.

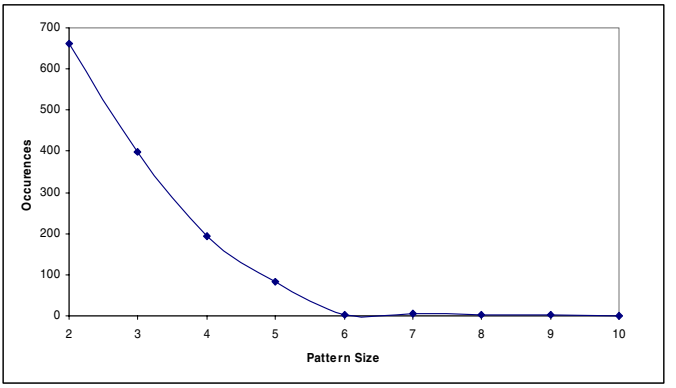


Fig. 4. Static occurrences and the pattern size of the unique patterns

3) Template Selection

In this step, a subset of templates is selected from the pattern library to reduce the complexity of the custom instruction selection process. This is necessary as the number of templates influences the computational complexity of the template matching process in custom instruction selection. In addition, restricting the number of templates can also lead to more efficient construction of morphable structures.

Although, custom instructions with small pattern sizes are likely to appear frequently in the embedded applications (see Fig. 4), templates with larger pattern sizes should also be selected as they can lead to significant speedup in certain applications. We employ a heuristic approach for template selection, which account for the performance gain and area utilization of the custom instruction in hardware. Each pattern p is assigned a gain as shown in (1). T_{SW} denotes the number of clock cycles taken for the custom instruction to run on a processor, and we assume each operation takes 1 clock cycle. T_{HW} denotes the number of clock cycles taken for the custom instruction in hardware, and we estimate this by the length of the critical path in the custom instruction sub-graph.

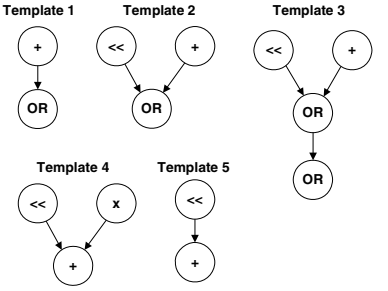
$$Gain(p) = \frac{T_{SW}\ (p)}{T_{HW} \times Pattern\ size\ (p)} \tag{1}$$

Patterns with higher gain values are selected as templates and stored in the template library. It is noteworthy that we do not consider the occurrences of the patterns in the selection decision as in [17]. This is because in our methodology, the templates are not used specifically for the standard application set. However, the pattern size of the custom instructions implicitly associates the occurrences of patterns in the gain as smaller patterns are likely to occur more frequently in embedded applications.

4.2 Identification and Characterization of Morphable Structures

The main task in this stage is to identify a unique set of morphable structures from the template library. Specifically, we aim to find a maximal unique set of

patterns that can cover all the templates. This is achieved by combining the larger sub-graphs in the template library, with smaller sub-graphs that are subsumed by it. The combination of subsumed sub-graphs is based on maximal similarity, which is defined as the minimal difference in the operations nodes of the two sub-graphs. Each pattern in the resulting maximal unique set cannot be subsumed by any other patterns in the set.



Template	Template with Maximal Similarity
1	2
2	3
3	None
4	None
5	4

Fig. 5. Identifying a maximal unique set of patterns from the templates

Fig. 5 illustrates the process of identifying a maximal unique set of patterns from the templates. In can be observed that although Template 1 can be subsumed by Template 2 and 3, it exhibit maximal similarity with Template 2. Hence, Template 1 is first combined with Template 2. Subsequently, Template 2 is combined with Template 3, and Template 5 is combined with Template 4. Finally, the remaining Templates 3 and 4 cannot be subsumed by each other and they formed the final set of maximal unique patterns. We can visually inspect that Templates 3 and 4 can cover Templates 1-5.

Combination of the subsumed patterns is equivalent to the sub-graph isomorphism problem. It is evident that this task is time consuming given the NP-completeness of the problem and the growing complexity of DFGs in modern embedded application. We have relied on the vflib graph-matching library [22] to find a maximal unique pattern set from the selected templates.

The morphable structures are then characterized to obtain their hardware performance and cost models to be used during custom instruction selection.

5 Experimental Results

In this section, we present experimental results to evaluate the benefits of our proposed methodology. We have selected a total of eight benchmarks from the MiBench embedded benchmark suite [8] as the standard application set. The baseline machine for the experiments is a four-wide VLIW architecture that can issue one integer, one floating-point, one memory, and one branch instruction each cycle. Table 1 shows the results obtained from the custom instruction identification process and pattern grouping. Although the pattern enumeration generates up to 1119 custom instruction instances, most of them can be grouped. After pattern grouping the number of unique patterns in the pattern library is reduced to 82 patterns.

Table 1. Results obtained from custom instruction identification and pattern grouping

Benchmarks	Custom instruction instances	Number of patterns in the pattern library
adpcm dec	17	
adpcm enc	22	
blowfish	990	
crc32	10	**82**
dijkstra	18	
FFT	6	
sha	34	
stringsearch	22	
Total	**1119**	

As can be observed from Table 1, a total of 82 templates can be used for custom instruction selection. Although it is desirable to limit the number of templates in order to increase the efficiency of template matching, we need to ensure that the resulting gain is not heavily compromised.

Fig. 6 shows the percentage cycles saved that can be achieved with varying number of templates used for template matching. The percentage cycles saved for application A is computed as shown in (2), where p_i for i = 1 to k, represent the k custom instructions selected for the application A, *dynamic occurrences*(p_i) is the execution frequency of the custom instruction p_i in application A, and *SW Clock Cycles*(A) denotes the number of clock cycles of the application A that is reported from Trimaran.

$$Percentage\ cycles\ saved(A) = \frac{\sum_{i=1}^{k} Clock\ cycles\ saved(p_i) \times dynamic\ occurences(p_i)}{SW\ Clock\ Cycles(A)} \times 100 \quad (2)$$

It can be observed that increasing the number of templates for matching will not lead to any notable gain after a certain point for each application. Hence, it is possible to reduce the number of templates for matching in order to achieve more efficient custom instruction selection, without compromising on the performance gain.

A total of 60 templates with highest gain values have been selected based on the approach described in Section 4.1.3. These templates consist of various pattern sizes (i.e. 2, 3, 4, 5, 6), which is necessary to accommodate to the different embedded applications. For example in Fig. 7, although the performance gain in most benchmark applications is contributed by small custom instructions (i.e. 2), larger custom instructions form a significant portion of the performance gain in certain benchmarks (i.e. sha).

Fig. 8 compares the performance obtained by the proposed technique with an approach based on application-centric template selection. We denote the latter as an application-centric approach. The application-centric approach performs pattern enumeration on each application individually to select templates using a gain that combines speedup and the pattern occurrences, which is similar to the approach

presented in [17]. In the application-centric approach, template matching is performed on the application using all the templates in the order of descending gain values. When a pattern match occurs, a custom instruction has been identified and the corresponding pattern is removed from the application DFG. The template matching process is repeated until there is no more pattern matches. It can be observed from Fig. 8 that the proposed method, which employs the same strategy for template matching (except that the gain in (1) is used and the number of templates are restricted to 60), provides comparable results with the application-centric approach. It is noteworthy that the proposed methodology executes much faster as it only performs the pattern enumeration process once. Moreover, as mentioned earlier, the employment of morphable structures dispenses the need for hardware syntheses flow in design exploration, and can give rise to area efficient implementations.

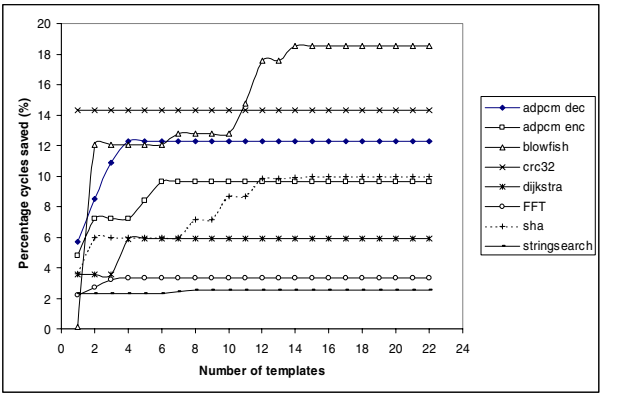


Fig. 6. Percentage cycles saved with varying number of selected templates used for template matching

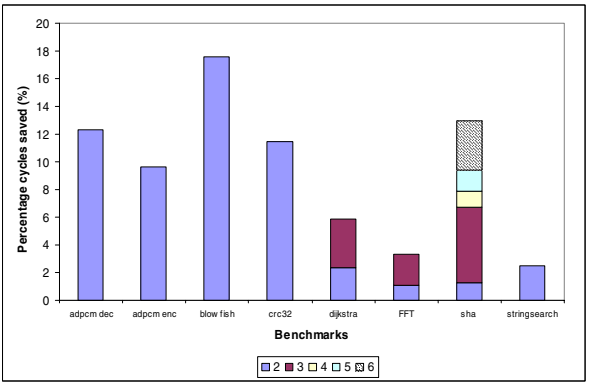


Fig. 7. Percentage cycles saved contributed by varying pattern sizes of the templates

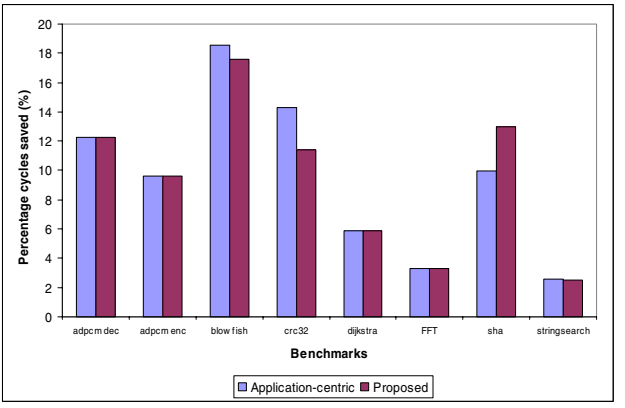


Fig. 8. Performance comparison of the proposed method with an application-centric approach

Table 2 shows the number of morphable structures required for each application and the total pattern sizes. As can be observed, the average number of morphable structures and the average number of operations for the eight applications is only 5.125 and 18.75 respectively. The maximum number of morphable structures is 8 with 45 operations, which is required by the sha application. These results imply that the reconfigurable area on the RISP can be kept small to cater to efficient custom instruction implementations.

Table 2. The number of morphable structures and the total pattern sizes for each application

Benchmarks	Required number of morphable structures	Sum of Pattern Sizes of the morphable structures
adpcm dec	5	19
adpcm enc	6	18
blowfish	6	21
crc32	2	8
dijkstra	5	14
FFT	6	15
sha	8	45
stringsearch	3	10
Average	**5.125**	**18.75**

6 Conclusion

We have proposed a methodology for instruction set customization on RISPs that uses morphable structures. The advantage of using morphable structures stems from the availability of a predefine set of morphable structures that can lead to rapid design exploration without a time-consuming hardware synthesis flow to evaluate the feasibility of the custom instruction candidates. In addition, the reconfigurable logic space to accommodate the morphable structures can also be minimized, as the number of morphable structures that are specific to a particular application is very small. The

experimental results show that 23 predefined morphable structures can sufficiently cater to any application in a set of eight MiBench benchmarks, and the average number of morphable structures per application is only 5.125 in order to achieve high performance gain. Future work includes validation of the methodology on a larger standard application set, and defining more effective criteria for the construction of morphable structures.

References

1. Dutt, N., Choi, K.: Configurable Processors for Embedded Computing, Computer, Vol. 36, No. 1, (2003) 120-123
2. Henkel, J.: Closing the SoC Design Gap, IEEE Computer, Vol. 36, No. 9 (2003) 119-121.
3. Xtensa Microprocessor: http://www.tensilica.com
4. ARCtangent Processor: http://www.arc.com
5. Barat, F., Lauwereins, R., Deconinck, G.: Reconfigurable Instruction Set Processors from a Hardware/Software Perspective, IEEE Transactions on Software Engineering, Vol. 28, No. 9 (2002) 847-862
6. Atasu, K., Pozzi, L., Ienne, P.: Automatic Application-Specific Instruction-Set Extensions Under Microarchitectural Constraints, Proceedings of the 40th IEEE/ACM Design Automation Conference (2003) 256-261
7. Ohlrich, M., Ebeling, C., Ginting, E., Sather, L.: SubGemini: Identifying Subcircuits Using a Fast Subgraph Isomorphism Algorithm, Proceedings of the 30th ACM/IEEE Design Automation Conference (1993) 31-37
8. Guthaus, M.R., Ringenberg, J.S., Ernst, D., Austin, T.M., Mudge, T., Brown, R.B.: MiBench: A Free, Commercially Representative Embedded Benchmark Suite, IEEE International Workshop on Workload Characterization (2001) 3-14
9. Wirthlin, M.J., Hutchings, B.L.: A Dynamic Instruction Set Computer, Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines (1995) 99-107
10. Kastrup, B., Bink, A., Hoogerbrugge, J.: ConCISe: A Compiler-Driven CPLD-based Instruction Set Accelerator, Proceedings.of the 7th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (1999) 92-101
11. Flaherty, N.: On the Chip or On the Fly, IEE Review, Vol. 50, No. 9 (2004) 48-51
12. Xilinx Platform FPGAs: http://www.xilinx.com
13. Altera Nios Soft Core Embedded Processor: http://www.altera.com
14. Kastner, R., Kaplan, A., Memik, S.O., Bozorgzadeh, E.: Instruction Generation for Hybrid Reconfigurable Systems, ACM Transactions on Design Automation of Embedded Systems, Vol. 7, No. 4, (2002) 605-627
15. Clark, N., Zhong, H., Mahlke, S.: Processor Acceleration Through Automated Instruction Set Customization, Proceedings of the 36th IEEE/ACM International Symposium on Microarchitecture (2003)
16. Biswas, P., Choudhary, V., Atasu, K., Pozzi, L., Ienne, P., Dutt, N.: Introduction of Local Memory Elements in Instruction Set Extensions, Proceedings of the 41st Annual IEEE/ACM Design Automation Conference, (2004) 729-734
17. Cong, J., Fan, Y., Han, G., Zhang, Z.: Application-Specific Instruction Generation for Configurable Processor Architectures, Proceedings of the 12th International Symposium on Field Programmable Gate Arrays (2004) 183-189
18. Sassone, P.G., Wills, D.S.: On the Extraction and Analysis of Prevalent Dataflow Patterns, Proceedings of the Workshop on Workload Characterization (2004)
19. Spadini, F., Fertig, M., Patel, S.: Charaterization of Repeating Dynamic Code Fragments, Technical Report CRHC-02-09, University of Illinois, Urbana-Champaign (2002)

20. Trimaran: An Infrastructure for Research in Instruction-Level Parallelism: http://www.trimaran.org
21. Yu, P., Mitra, T.: Characterizing Embedded Applications for Instruction-Set Extensible Processors, Proceedings of the 41st IEEE/ACM on Design Automation Conference (2004) 723-728
22. Cordella, L.P., Foggia, P., Sansone, C., Vento, M.: Performance Evaluation of the VF Graph Matching Algorithm, International Conference on Image Analysis and Processing (1999) 1172-1177

Implementation of a Hybrid TCP/IP Offload Engine Prototype*

Hankook Jang, Sang-Hwa Chung, and Soo-Cheol Oh

Department of Computer Engineering,
Pusan National University,
Pusan, 609-735, Korea
{hkjang, shchung, osc}@pusan.ac.kr

Abstract. Recently TCP/IP Offload Engine (TOE) technology, which processes TCP/IP on a network adapter instead of the host CPU, has become an important approach to reduce TCP/IP processing overhead in the host CPU. There have been two approaches to implementing TOE: software TOE, in which TCP/IP is processed by an embedded processor on a network adapter; and hardware TOE, in which all TCP/IP functions are implemented by hardware. This paper proposes a hybrid TOE that combines software and hardware functions in the TOE. In the hybrid TOE, functions that cannot have guaranteed performance on an embedded processor because of heavy load are implemented by hardware. Other functions that do not impose as much load are implemented by software on embedded processors. The hybrid TOE guarantees network performance near that of hardware TOE and it has the advantage of flexibility, because it is easy to add new functions or offload upper-level protocols of TCP/IP. In this paper, we developed a prototype board with an FPGA and an ARM processor to implement a hybrid TOE prototype. We implemented the hardware modules on the FPGA and the software modules on the ARM processor. We also developed a coprocessing mechanism between the hardware and software modules. Experimental results proved that the hybrid TOE prototype can greatly reduce the load on a host CPU and we analyzed the effects of the coprocessing mechanism. Finally, we analyzed important features that are required to implement a complete hybrid TOE and we predict its performance.

1 Introduction

Ethernet technology has been widely used in many areas such as the Internet. It has already achieved a bandwidth of 1 Gbps, and 10-Gigabit Ethernet is being adopted in some areas. TCP/IP, the most popular communication protocol for Ethernet, is processed on a host CPU in most computer systems. This imposes a heavy load on the host CPU, thus degrading the overall performance of computer

* This work was supported by the Regional Research Centers Program(Research Center for Logistics Information Technology), granted by the Korean Ministry of Education & Human Resources Development.

T. Srikanthan et al. (Eds.): ACSAC 2005, LNCS 3740, pp. 464–477, 2005.

systems. At the same time, more and more load is imposed on a host CPU as the physical bandwidth of the network increases [1,2]. To solve this problem, the TCP/IP Offload Engine (TOE), in which TCP/IP is processed on a network adapter instead of a host CPU, has been introduced. A TOE can greatly reduce the load on a host CPU and help the CPU concentrate on executing computation jobs rather than communication jobs, thus improving the overall performance of the computer system.

There have been two approaches in developing TOE. In the first approach, an embedded processor installed on a network adapter processes TCP/IP using software (software TOE) [3]. The software TOE has the advantage that it is easier to implement than implementing all TCP/IP functions as hardware. However, this approach has a disadvantage in network performance [4] that an embedded processor generally has poor performance compared with a host CPU and processing TCP/IP on the embedded processor is slow. The other approach is to develop a specialized ASIC processing TCP/IP (hardware TOE) [5,6,7]. The hardware TOE guarantees network performance [8,9,10], but this approach has the disadvantage of inflexibility because it is difficult to add new functions. Moreover, in a network adapter based on hardware TOE, it is difficult to process upper-level protocols of TCP/IP.

In our previous study [11], we analyzed important factors that impose high loads on a host CPU by measuring times spent in processing each function in the Linux TCP/IP protocol stack. Based on these analyses, we proposed a hybrid TOE architecture that combines hardware TOE and software TOE. In hybrid TOE, functions that cannot have guaranteed performance on an embedded processor because of heavy load are implemented by hardware. Other functions, such as connection establishment, which do not impose as much load, are processed by software on embedded processors. The hybrid TOE guarantees network performance by implementing compute-intensive functions as hardware. It also has the advantage of flexibility because it is easy to add new functions and offload upper-level protocols based on TCP/IP.

For this paper, we developed a prototype board that has an FPGA and an ARM processor to implement a hybrid TOE prototype. This board is connected to the host CPU through the 64-bit/66-MHz PCI bus. Hardware modules for the hybrid TOE prototype were implemented on the FPGA and software modules for it were implemented in the ARM processor. We also developed a coprocessing mechanism between hardware modules and software modules. The experimental results showed that the hybrid TOE prototype can greatly reduce the load on a host CPU and we analyzed the effects of the coprocessing mechanism. Finally, we analyzed some features that are required to implement a complete hybrid TOE and we predict its performance.

This paper is organized as follows. In Section 2, related works are explained. In Section 3, the hybrid TOE architecture is proposed. In Section 4, we explain the implementation of the hybrid TOE prototype. In Section 5, experimental results are shown and analyzed. Finally, we present conclusions and future work in Section 6.

2 Related Works

Between two traditional approaches in developing TOE, Intel PRO/1000T IP Storage Adapter [3] is an example of Software TOE product. This is based on a Gigabit Ethernet adapter and equipped with an Intel 80200 StrongARM processor (200 MHz). In this adapter, TCP/IP and iSCSI(SCSI over IP) protocol are processed by the processor. According to experiments performed at Colorado University [4], unidirectional bandwidth of this adapter does not exceed 30 MB/s, which is about half the bandwidth of 70 MB/s achieved by general Gigabit Ethernet adapters. This shows that Software TOE has a disadvantage in network performance.

Alacritech's SLIC [5], Adaptec's NAC-7711 [6], and QLogic's ISP4010 [7] are examples of Hardware TOE products. All of these are supporting Gigabit Ethernet. According to benchmark reports, unidirectional bandwidths of Hardware TOE products are about 100 MB/s [8,9,10] that is near the peak bandwidth of Gigabit Ethernet. This shows that Hardware TOE guarantees network performance.

3 Hybrid TOE Architecture

Fig. 1 shows the structure of a hybrid TOE adapter based on the hybrid TOE architecture proposed in this paper. The hybrid TOE adapter is equipped with a hybrid TOE module, a TOE interface, a memory controller, and a Gigabit Ethernet controller. The hybrid TOE module, which is the most important part of the hybrid TOE adapter, consists of two embedded processors and a TOE hardware module.

In the hybrid TOE, TCP/IP processing work is divided into transmission work and reception work, and the two tasks are processed by the embedded processors. This mechanism helps the software modules in the hybrid TOE overcome the performance limitations of a single embedded processor that has lower performance than a host CPU. Moreover, scheduling overheads imposed by task switching between transmission and reception work will be reduced. The TOE hardware module processes TCP/IP functions implemented by hardware. The TOE interface connects the hybrid TOE module and the host CPU. The memory controller is responsible for managing buffer memory for packet buffers, which consist of a header area for storing TCP/IP/MAC headers and a data area. The Gigabit Ethernet controller is responsible for controlling the Gigabit Ethernet MAC/PHY chipset.

When a user program requests a communication using the hybrid TOE adapter, this request is delivered to the TOE interface. The hybrid TOE module then reads the request from the TOE interface and performs operations for the request using a hardware/software coprocessing mechanism. If the request is for data transmission, the hybrid TOE module creates a packet buffer, and then copies data from the main memory of the host CPU into the data area of the packet buffer using DMA. The hybrid TOE module then completes generating

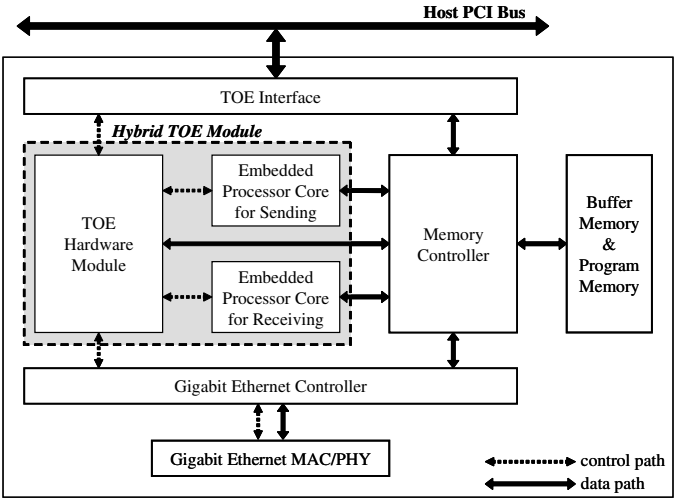


Fig. 1. Structure of the hybrid TOE adapter

the packet by creating the TCP, IP and MAC headers in the header area of the packet buffer. After generating the packet, the hybrid TOE module requests the Gigabit Ethernet controller to transmit the packet. The Gigabit Ethernet MAC fetches the packet from the packet buffer using DMA and transmits it to a receive node. At the receive node, the incoming packet is processed in the reverse order of transmission processing flow. After the request is processed at the hybrid TOE adapter, the TOE interface interrupts the host CPU to report the completion of request processing. Then the host CPU fetches the result of the request processing from the TOE interface and delivers it to the user program.

The main functions of TCP/IP are (1) connection establishment, (2) packet buffer creation and freeing, (3) TCP header creation and processing, (4) acknowledgement (ACK) packet creation and processing, and (5) flow control. According to analyses of Linux's TCP/IP protocol stack [11], about 70∼90 % of TCP/IP processing time is spent in tasks (2), (3), (4) and task scheduling, so these functions must be optimized for implementing TOE. In the hybrid TOE, tasks (2), (3), and (4) are implemented by hardware, and other functions as well as task scheduling are implemented by software on embedded processors.

4 Implementation of the Hybrid TOE Prototype

4.1 Hybrid TOE Prototype Board

For this paper, we developed a hybrid TOE prototype board that is equipped with an FPGA and an embedded processor as a step toward implementing a complete hybrid TOE. We then developed a hybrid TOE prototype using the prototype board. Fig. 2 shows the structure of the hybrid TOE prototype board.

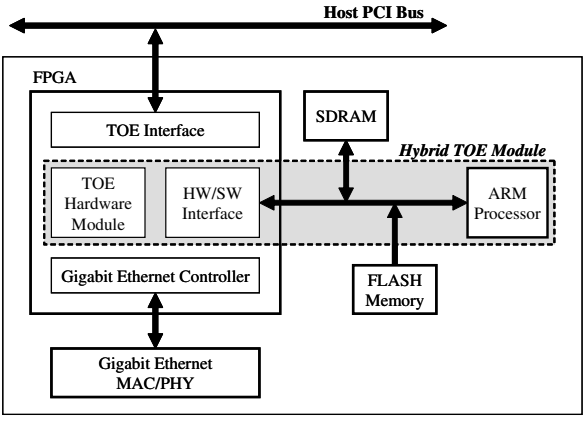


Fig. 2. Block diagram of the hybrid TOE prototype board

This board is equipped with an FPGA, an embedded processor, a Gigabit Ethernet MAC/PHY chipset, an SDRAM module and a flash memory. We adopted a Xilinx Virtex-II Pro FPGA (XC2VP30) for implementing hardware modules of the hybrid TOE prototype and a Samsung ARM processor (S3C2410X) for implementing software modules of the hybrid TOE prototype. The National DP82820/DP83865 chipset was adopted for the Gigabit Ethernet MAC/PHY. The sizes of SDRAM and flash memory are 64 MB and 16 MB, respectively.

The TOE interface, TOE hardware module, HW/SW interface, and Gigabit Ethernet controller were implemented in the FPGA. The TOE interface connects the host CPU and hybrid TOE module. The TOE hardware module consists of three hardware units that correspond to using packet buffers on data transmission. The HW/SW interface is a key element for hardware/software coprocessing, and provides an interface between the TOE hardware module and the ARM processor. The Gigabit Ethernet controller is responsible for controlling the Gigabit Ethernet MAC/PHY chipset.

The hybrid TOE prototype board and the host CPU are connected by the 64-bit/66-MHz PCI bus and the hardware modules in the FPGA are driven by the 66 MHz PCI clock. The ARM processor operates with a 135 MHz core clock for the ARM core and a 67.5 MHz system clock for the memory bus. The system clock is used by the ARM processor to access SDRAM, flash memory, and the FPGA.

4.2 TOE Interface and Protocol Stack for Hybrid TOE

Fig. 3 shows the structure of the protocol stack based on the hybrid TOE prototype and TOE interface. The BSD socket layer in this stack is directly connected to the device driver of the hybrid TOE prototype board by bypassing the TCP/IP layer. The device driver is connected to the TOE interface of the hybrid TOE prototype through the host PCI bus. The TOE interface provides the device driver with a socket-based interface.

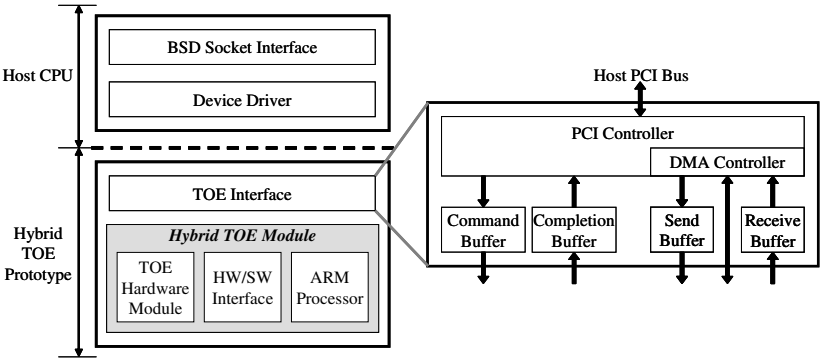


Fig. 3. Structure of the protocol stack based on the hybrid TOE prototype

The TOE interface consists of a PCI controller, command buffer, completion buffer, send buffer, and receive buffer. A command request corresponding to a socket function in the BSD socket layer is delivered to the hybrid TOE module through the command buffer. After a command request is processed by the hybrid TOE prototype, a completion result reporting the result of request processing is delivered to the host CPU through the completion buffer. The send buffer is used to store the data that will be copied to SDRAM by the ARM processor, and the data in the main memory is fetched to the send buffer by DMA. The receive buffer is used to store received data that will be transferred into the main memory by DMA.

This protocol stack operates as follows. When a user program invokes a function of the BSD socket layer, the device driver is invoked directly by the BSD socket interface. The device driver creates a command request corresponding to the request from the user program and stores it in the command buffer. The hybrid TOE module then reads the command request and performs operations corresponding to the request. After the request is processed, the hybrid TOE module creates the completion result and stores it in the completion buffer. The TOE interface then interrupts the host CPU to report the completion of request processing. The host CPU reads the completion result and reports the result to the user program.

4.3 TOE Software Module

The TOE software module of the hybrid TOE prototype was developed using embedded Linux on the ARM processor. We chose this platform because it has higher network performance than other operating systems and it is easy to modify the kernel including the TCP/IP module. Moreover, embedded Linux can provide a convenient environment for developing upper-level protocols based on the hybrid TOE.

Fig. 4 shows the structure of the TOE software module. TOE agents take charge of performing communications and a TOE manager is responsible for

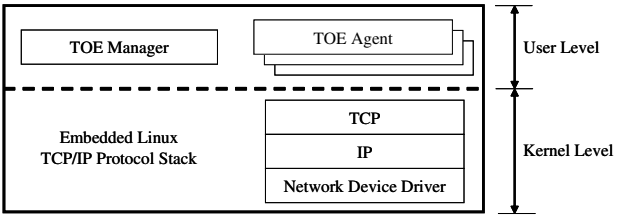


Fig. 4. Structure of the TOE software module

managing the TOE agents. These are user-level threads and perform communication by reading a command request and invoking a socket function corresponding to the request. Each TOE agent is created in the ARM processor whenever the host CPU creates a socket for a TCP connection. The TOE manager manages creation and termination of TOE agents, and is responsible for invoking the corresponding TOE agent when the host CPU transmits or receives data using the hybrid TOE prototype. The network device driver supports the Gigabit Ethernet controller.

The TOE software module processes a data transmission as follows. When a command request for data transmission is stored in the command buffer in the TOE interface, the HW/SW interface detects it and interrupts the ARM processor. In the ARM processor, an interrupt handler invokes the TOE manager, and then the TOE manager creates a TOE agent that will process the command request. The TOE agent invokes the send function of the TCP/IP protocol stack. In the TOE software module, the interrupt handler operates in the kernel level, but the TOE manager and TOE agents operate in the user level.

After the TOE agent invokes the send function, socket buffers are first created in the TCP layer. A socket buffer requires memory to store the data and the TCP, IP, and MAC headers. For this memory space, a packet buffer provided by the TOE hardware module is used. (The TOE hardware module will be explained in detail in Section 4.4). After the data in the main memory have been copied into the data area of the packet buffer using DMA, the TOE software module creates the TCP, IP and MAC headers in SDRAM. The TOE software module completes the packet creation by copying these headers into the header area of the packet buffer. After the packet creation, the Gigabit Ethernet controller requests the Gigabit Ethernet MAC to transmit the packet.

The TOE software module processes an incoming packet as follows. After the packet is stored in the receive (RX) buffer in the Gigabit Ethernet controller, the controller interrupts the ARM processor to report packet reception. In the ARM processor, the device driver creates a socket buffer with a packet buffer. The device driver then copies the packet in the RX buffer into the packet buffer of the socket buffer. After TCP/IP reception processing, the TOE agent copies the received data into the receive buffer in the TOE interface. Then the TOE interface transfers the data to a user buffer in the main memory by DMA and interrupts the host CPU, thus completing reception.

4.4 TOE Hardware Module and Coprocessing Mechanism

Fig. 5 shows the structure of the TOE hardware module and HW/SW interface. For this paper, we developed a TOE hardware module that consists of three hardware units related to packet buffers and data transmission. These units are a Memory Allocation Unit (MAU), Data Fetch Unit (DFU) and Partial Checksum Calculation Unit (PCU). We also developed a HW/SW interface to support the hardware/software coprocessing mechanism that allows the TOE hardware module and the ARM processor to share information for TCP/IP processing through the HW/SW interface.

The MAU in the TOE hardware module manages packet buffer memory to create packet buffers, which are used for storing headers as well as data. The MAU divides the packet buffer memory into small slots to allocate a memory space as quickly as possible when creating a packet buffer. Each slot is 2048 bytes, which is enough to store an Ethernet MTU of 1514 bytes plus other control fields of the Linux socket buffer structure. After a packet buffer is created, the DFU fetches data from the main memory using the DMA controller in the TOE interface and stores it in the data area of a packet buffer without using the send buffer in the TOE interface. While the DFU copies data, the PCU calculates a partial checksum of the data. After a TCP header is generated, the partial checksum is used for calculating the TCP checksum, which is the sum of the partial checksum of the data and a checksum of the TCP header. The time taken for the PCU to calculate a partial checksum does not affect the communication latency because the PCU operates in parallel with the DFU.

In the HW/SW interface, the ADDRESS field is used to store the address of a packet buffer that was allocated by MAU. The READY field is used to inform the ARM processor when a packet buffer is ready. The DMA_DONE field

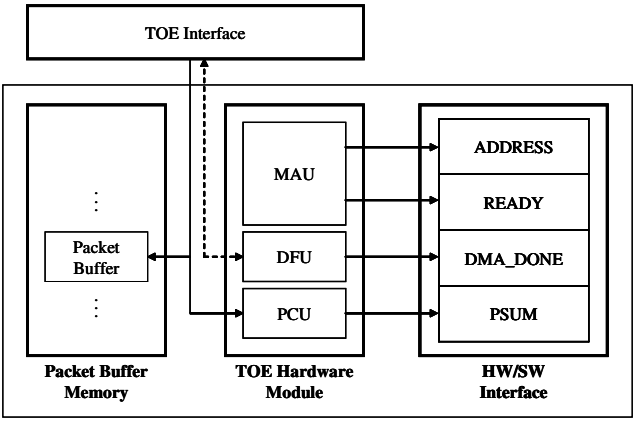


Fig. 5. Structure of the TOE hardware module and HW/SW interface

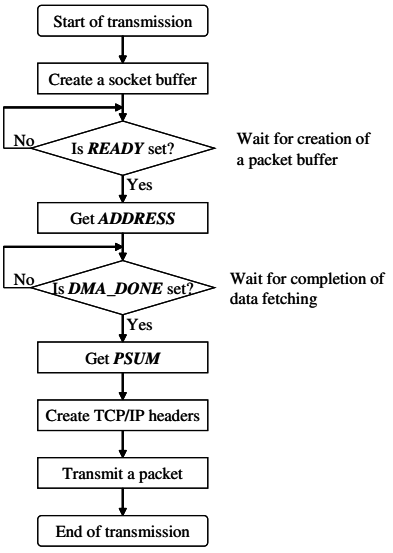


Fig. 6. Flow of data transmission in the ARM processor

informs the ARM processor whether the DMA for data copying is complete. The partial checksum of the data is stored in the PSUM field after the DMA is complete.

Fig. 6 shows a flow chart of data transmission in the ARM processor, which uses the READY and the DMA_DONE fields in the HW/SW interface for hardware and software coprocessing. When creating a socket buffer, the ARM processor checks the READY field by polling until the MAU allocates a packet buffer and sets the READY field. After the READY field is set, the ARM processor reads the ADDRESS field and obtains the address of the packet buffer that was allocated by the MAU. The ARM processor then waits for completion of the data copy and partial checksum calculation by polling the DMA_DONE field. After the DMA_DONE field is set, the ARM processor reads the partial checksum and creates the TCP/IP and MAC headers in SDRAM. The ARM processor completes the packet creation by copying these headers to the header area of the packet buffer, and the packet is then transmitted.

5 Experiments and Analyses

For this paper, we measured the performance of the hybrid TOE prototype and analyzed important features required for developing the complete hybrid TOE. In the experiments, two nodes were connected using hybrid TOE prototype boards without a switch. Each node had a 1.8 GHz Intel Xeon CPU, 512 MB of main memory and 64-bit/66-MHz PCI bus. The operating system on the host CPU was based on Linux kernel 2.4.7-10.

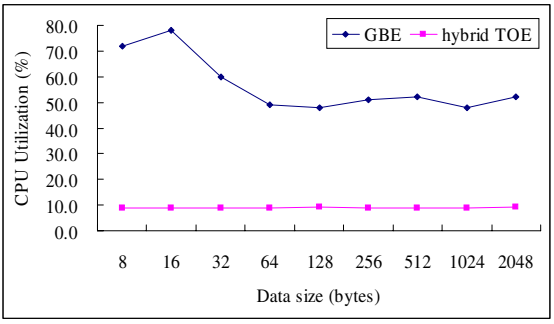


Fig. 7. CPU utilization of GBE and TOE

5.1 CPU Utilization

We compared the CPU utilization of the hybrid TOE prototype (hybrid TOE) with that of a Gigabit Ethernet adapter (GBE), using TCP/IP communication. The Gigabit Ethernet adapter used was the Intel PRO/1000MT Server Adapter. As shown in Fig. 7, the CPU utilization with the GBE varied from about 50 % to 80 % according to data size. However, the CPU utilization of the TOE was always about 9 %, much lower than that of the GBE. This is mainly because the TCP/IP protocol is processed in the hybrid TOE prototype. This result shows that the hybrid TOE prototype greatly reduces the load on the host CPU.

5.2 Comparison of Hybrid TOE and Software TOE

We compared the performance of the hybrid TOE prototype (hybrid TOE) with that of the software TOE to show the effects of hardware/software coprocessing. In the software TOE, all TCP/IP functions including the three hardware units of the hybrid TOE are processed in the ARM processor using software. For more detailed analysis, we measured times spent in processing data transmission through TCP, IP and the device driver layer in the hybrid TOE and the software TOE, as shown in Table 1. Performance improvement of the hybrid TOE was achieved when transmitting data because the three hardware units of the hybrid TOE and the hardware/software coprocessing mechanism were used.

The main features causing performance improvement are create socket buffer (S2), fetch data (S3) and copy packet (S8) in Table 1. The time required to create a socket buffer in the hybrid TOE is 21.2 μs, 9 μs faster than for the software TOE. This is because the packet buffer creation, included in the socket buffer creation, is implemented by hardware and the time taken to create a packet buffer is reduced to 1 μs from 10 μs. The performance improvements for fetch data (S3) and copy packet (S8) result from reduction of data copies. Packet buffers for the hybrid TOE are maintained in the FPGA, but packet buffers for software TOE are in SDRAM. In the software TOE, data in the main memory is copied to the packet buffer in SDRAM through a send buffer in the TOE interface and the created packet is copied again into the transmission (TX) buffer in the

Table 1. Time spent in sending data (time unit : μs)

Label	Operation	Hybrid TOE	Software TOE
S1	Enter TCP layer from interrupt handler	469.2	469.2
S2	Create socket buffer (Create packet buffer)	21.2 (1.0)	30.2 (10.0)
S3	Fetch data from main memory	7.7	69.6
S4	Create TCP header	48.9	48.9
S5	Enter IP layer from TCP layer	52.6	52.6
S6	Create IP header	49.0	49.0
S7	Enter device driver from IP layer and Create MAC header	32.7	32.7
S8	Copy packet to TX Buffer	12.2	690.5
S9	Complete transmission	104.2	104.2
	Total transmission time	797.8	1547.4

Gigabit Ethernet controller. The copied packet in the TX buffer is delivered to a receive node by the Gigabit Ethernet MAC. In contrast, two extra data copies to and from SDRAM are not necessary in the hybrid TOE prototype, because it maintains packet buffers in the FPGA and this results in the performance improvement.

5.3 Performance Prediction

Based on the experimental results of Section 5.2, we analyzed important features for implementing a complete hybrid TOE and we predict the performance of the complete hybrid TOE, in which all features of the hybrid TOE are completely implemented. In Table 2, features F1–F3 will be implemented by hardware and F4–F5 will be optimized using software based on an ARM processor. These features are for data transmission and similar features will be adopted for data reception in the complete hybrid TOE.

As shown in Table 1, the time spent in creating a packet buffer was 10 μs when using the software TOE, but it was reduced to 1 μs by adopting hardware units for the hybrid TOE prototype. In the hybrid TOE prototype, it takes about 70 clocks for the MAU to obtain a free slot from the packet buffer memory and set the ADDRESS and READY fields in the HW/SW interface. All hardware modules in the FPGA are operated with the 66 MHz PCI clock, so 70 clocks are near 1 μs. In the complete hybrid TOE, we will improve the MAU to allocate memory spaces not just for the packet buffers but for the socket buffers (F1); then it will take about 1 μs to create a socket buffer, similar to the time for packet buffer creation in the hybrid TOE prototype.

In the complete hybrid TOE, the TCP, IP, and MAC headers will be created by hardware (F2), and we predict the time required to create headers as follows. To create each header by hardware, three operations are required. These

Table 2. Features required to implement complete Hybrid TOE (time unit : μs)

Label	Features	Corresponding terms in Table 1	Processing time on hybrid TOE prototype	Processing time on complete hybrid TOE
F1	Manage socket buffer by hardware	S2	21.2	1.0
F2	Create TCP/IP/MAC headers by hardware	S4	48.9	1.5
		S6	49.0	1.5
		S7	32.7	1.5
		S8	12.2	0.0
F3	Optimize completion flow	S9	104.2	10.0
F4	Enter TCP layer directly from interrupt handler	S1	469.2	3.6
F5	Remove overhead on entering every layer	S5	52.6	3.6
		S7	32.7	3.6
	Total processing time		822.7	26.3

operations take the corresponding information from memory for managing TCP connections, fill every field of a temporary header buffer, and copy the header in the temporary header buffer to the header area of the packet buffer. This processing sequence is similar to that of the MAU, so we can expect that it will take less than 100 clocks to create each header and it will therefore take about 300 clocks (4.5 μs) to create all headers. In addition, the overhead of copying headers from SDRAM to the TX buffer (S8 in Table 1) will be removed when creating headers in the packet buffer by hardware.

For the complete hybrid TOE, we will also implement a hardware unit that creates a completion result in the TOE interface (F3). We expect that it will take about 1 μs to create a completion result similar to the processing time of the MAU. It took at most 9 μs in our experiments for the host CPU to read a completion result after an interrupt, so it will take about 10 μs to report a completion result.

When a command request arrives at the command buffer, it is delivered to the TCP/IP protocol stack of the TOE software module through the interrupt handler, the TOE manager, and a TOE agent. This sequence requires a long processing time because these three modules invoke one another and each transition is performed by the Linux scheduler. For the complete hybrid TOE, we will optimize this sequence so that the interrupt handler can directly invoke a function of the TCP/IP protocol stack instead of using the Linux scheduler (F4). In our experiments, it took about 3.6 μs for the interrupt handler to invoke a function in the device driver. Thus the time taken to process F4 will be about 3.6 μs, similar to the experimental result.

For the complete hybrid TOE, we will also remove unnecessary functions and scheduling overhead between each protocol layer (F5). We expect that the time

spent in transition from one layer to the next layer will be about 3.6 μs, as for F4. Thus it will take about 7.2 μs for the two transitions S6 and S8 in Table 1.

In the complete hybrid TOE, reception sequences will also be optimized by hardware implementation, and we predict the latency of the complete hybrid TOE as follows. When transmitting a 1024-byte data packet, it will take about 34 μs, which is the sum of 7.7 μs (S3 in Table 1) and 26.3 μs (processing time in Table 2, to process data transmission. If data reception is optimized similarly to data transmission, it will take less than 70 μs to process both transmission and reception. In our experiments, about 40 μs was spent in creating a command request, storing the command request to the command buffer, and propagating data through Gigabit Ethernet network. This delay will not be reduced in the complete hybrid TOE. Thus, the latency of complete hybrid TOE, for a 1024-byte data packet, will be about 110 μs. This latency is less than the 120 μs of a Gigabit Ethernet adapter [10] and more than the 90 μs of Adaptec's NAC-7711 [10], which is a hardware TOE product.

6 Conclusions and Future Work

In this paper, we proposed a hybrid TOE architecture and we developed a hybrid TOE prototype equipped with an FPGA and an ARM processor. To support user programs in using the hybrid TOE prototype, we developed a protocol stack on a host CPU, in which the socket layer is connected to the device driver layer by bypassing TCP/IP. In the hybrid TOE prototype, we developed a TOE interface to the host CPU and we implemented it in the FPGA. We implemented hardware modules for the hybrid TOE prototype using the FPGA and software modules for the hybrid TOE prototype using the ARM processor. We also developed a coprocessing mechanism between hardware modules and software modules, and then we implemented a HW/SW interface for the coprocessing mechanism on the FPGA. Experimental results showed that the host CPU utilization is about 9 % when using the hybrid TOE prototype and greater than 50 % when using a general Gigabit Ethernet adapter. This proves the benefit of the hybrid TOE prototype in greatly reducing the load on the host CPU. Another result showed that the hybrid TOE prototype based on a hardware/software coprocessing mechanism outperforms a software TOE. Moreover, we analyzed essential features for implementing a complete hybrid TOE. Based on the analyses, we predict that the complete hybrid TOE will have a one-way latency of 113 μs when transmitting 1024-byte data packets. This latency is near the latencies of hardware TOE products and general Gigabit Ethernet adapters. In future work, we will develop a complete hybrid TOE by implementing all the features analyzed in this paper and adopting faster processors than the ARM processor.

References

1. Bierbaum, N.: MPI and Embedded TCP/IP Gigabit Ethernet Cluster Computing. Proceedings of 27th Annual IEEE Conference on Local Computer Networks 2002 (2002) 733–734
2. Yeh, E., Chao, H., Mannem, V., Gervais, J., Booth, B.: Introduction to TCP/IP Offload Engine (TOE). 10 Gigabit Ethernet Alliance (2002)
3. Intel Corporation: Intel PRO/1000T IP Storage Adapter. Data Sheet (2003) http://www.intel.com/network/connectivity/resources/doc_library/data_sheets/pro1000_T_IP_SA.pdf
4. Aiken, S., Grunwald, D., Pleszkun, A.R., Willeke, J.: A Performance Analysis of the iSCSI Protocol. Proceedings of the 20th IEEE/11th NASA Goddard Conference on Mass Storage Systems and Technologies (2003)
5. Alacritech, Inc.: SLIC Technology Overview. Technical Review (2002) http://www.alacritech.com/html/tech_review.html
6. Adaptec, Inc.: Adaptec TOE NAC 7711. Data Sheet (2003) http://graphics.adaptec.com/pdfs/ana_7711_datasheet.pdf
7. QLogic Corporation: iSCSI Controller, Data Sheet (2003) http://download.qlogic.com/datasheet/16291/isp4010.pdf
8. Lionbridge Technologies, Inc.: Alacritech SES1001T: iSCSI HBA Competitive Analysis. VeriTest Benchmark Report (2004) http://www.veritest.com/clients/reports/alacritech/alac_ses1001t.pdf
9. Adaptec, Inc.: Unleashing File Server Potential with Adaptec GigE NAC 7711. Benchmark Report (2003) http://graphics.adaptec.com/pdfs/NAC_appbrief.pdf
10. Ghadia, H.: Benefits of full TCP/IP offload (TOE) for NFS Services. Proceedings of 2003 NFS Industry Conference (2003) http://nfsconf.com/pres03/adaptec.pdf
11. Oh, S.-C., Jang, H., Chung, S.-H.: Analysis of TCP/IP protocol stack for a Hybrid TCP/IP Offload Engine. Proceedings of the 5th International Conference on Parallel and Distributed Computing, Applications and Technologies (2004) 406–409

Matrix-Star Graphs: A New Interconnection Network Based on Matrix Operations

Hyeong-Ok Lee[1], Jong-Seok Kim[2], Kyoung-Wook Park[3], Jeonghyun Seo[4], and Eunseuk Oh[5]

[1] Department of Computer Education, Sunchon National University, 315 Maegok-dong, Sunchon, Chonnam, 540-742, Korea
oklee@sunchon.ac.kr
[2] Department of Computer Science, Oklahoma State University, USA
[3] Department of Computer Science, Chonnam National University, Korea
[4] Department of Computer Science, Sunchon National University, Korea
[5] Department of Computer Science, Texas A&M University, USA

Abstract. In this paper, we introduce new interconnection networks *matrix-star graphs* $MTS_{n_1,\ldots,n_k}$ where a node is represented by $n_1 \times \ldots \times n_k$ matrix and an edge is defined by using matrix operations. A matrix-star graph $MTS_{2,n}$ can be viewed as a generalization of the well-known star graph such as degree, connectivity, scalability, routing, diameter, and broadcasting. Next, we generalize $MTS_{2,n}$ to 2-dimensional and 3-dimensional matrix-star graphs $MTS_{k,n}$, $MTS_{k,n,p}$. One of important desirable properties of interconnection networks is network cost which is defined by degree *times* diameter. The star graph, which is one of popular interconnection topologies, has smaller network cost than other networks. Recently introduced network, the macro-star graph has smaller network cost than the star graph. We further improve network cost of the macro-star graph: Comparing a matrix-star graph $MTS_{k,k,k}(k = \sqrt[3]{n^2})$ with $n^2!$ nodes to a macro-star graph $MS(n-1, n-1)$ with $((n-1)^2+1)!$ nodes, network cost of $MTS_{k,k,k}$ is $O(n^{2.7})$ and that of $MS(n-1, n-1)$ is $O(n^3)$. It means that a matrix-star graph is better than a star graph and a macro-star graph in terms of network cost.

Keywords: Interconnection network, Star Graph, Macro-star Graph.

1 Introduction

An interconnection network can be represented as an undirected graph $G(V, E)$ where a processor is represented as a node $u \in V(G)$, and a communication channel between processors as an edge $(u, v) \in E(G)$ between corresponding nodes u and v. Popular interconnection networks include mesh, hypercube[7], and star graph[1]. The most widely used criteria to evaluate interconnection networks are degree, connectivity, scalability, diameter, fault tolerance, symmetry[1,3,7]. There is a trade-off between degree, related to hardware cost, and diameter, related to transmission time of messages. In general, the network throughput increases as degree of network increases, but hardware cost is also getting higher

T. Srikanthan et al. (Eds.): ACSAC 2005, LNCS 3740, pp. 478–487, 2005.

due to the more number of pins of processors connected. On the other hand, a network with smaller degree can reduce hardware cost, but latency time or throughput is poor because delay of message transmission increases. Thus, network cost, defined by degree × diameter, has been introduced to evaluate desirable properties of interconnection networks[2,8]. When interconnection networks have the same number of nodes, the network with a smaller cost of degree × diameter is regarded as more desirable.

An n-dimensional star graph S_n consists of nodes represented by permutations of n symbols, and nodes are connected if their permutations are obtained by exchanging the first symbol and other symbol. S_n has $n!$ nodes, $\frac{(n-1)n!}{2}$ edges, degree of $n-1$, diameter of $\lfloor \frac{3(n-1)}{2} \rfloor$, and its network cost is approximately $\frac{3n^2}{2} - 3n$. S_n has node and edge symmetry while it has smaller degree and diameter than the hypercube. However, embedding other networks into S_n is rather complicated and many nodes are not equally loaded. Expanding from S_n to S_{n+1} is also impractical because a large number of nodes must be added[1]. To overcome such shortcomings, its variations such as the bubblesort star graphs[4], the transposition graphs[6], the star connected cycles[5], and the macro-star graphs[9] have been introduced. A bubblesort star graph is a graph generated by merging star graphs. A transposition graph was developed to reconstruct a star graph by using alternative edges when it has faulty edges. Star connected cycles fixed its degree to 3 by substituting a ring for each node of a star graph. A macro-star graph reduced the degree of a star graph by half, and it is the best known graph as an alternative to the star graph in terms of network cost.

In this paper, we introduce a new interconnection network to further improve the network cost of a macro-star graph, while it holds attractive properties of the star graph such as scalability and maximum fault-tolerance. We demonstrate desirable properties of the matrix-star graph as an interconnection network in the following sections. In section 2, we define a $2 \times n$ matrix-star graph $MTS_{2,n}$, and show that it has scalability and maximum fault-tolerance. In section 3, we develop a routing algorithm and analyze the diameter of the matrix-star graph. In section 4, we develop a broadcasting algorithm. In section 5, we generalize $MTS_{2,n}$ to a $k \times n$ matrix-star graph $MTS_{k,n}$ and a $k \times n \times p$ matrix-star graph $MTS_{k,n,p}$. Finally, we conclude the paper in section 6.

2 Topological Properties of Matrix-Star Graphs

Let $MTS_{2,n}$ be a *Matrix-Star graph*, where a node is represented by a $2 \times n$ matrix $\begin{bmatrix} x_1 & x_2 & \dots & x_i & \dots & x_n \\ x_{n+1} & x_{n+2} & \dots & x_j & \dots & x_{2n} \end{bmatrix}$ consisting of $2n$ symbols $\{1, 2, ..., 2n\}$, and two nodes are connected if and only if a node is obtained by the matrix operations $\mathcal{C}$, $\mathcal{E}$, and $\mathcal{R}$ from the other node as follows: for a node $u = \begin{bmatrix} x_1 & x_2 & \dots & x_i & \dots & x_n \\ x_{n+1} & x_{n+2} & \dots & x_j & \dots & x_{2n} \end{bmatrix}$, (1) the first symbol in the first row is exchanged with the ith symbol in the first row

$$\mathcal{C}_i(u) = \begin{bmatrix} x_i & x_2 & \dots & x_1 & \dots & x_n \\ x_{n+1} & x_{n+2} & \dots & x_j & \dots & x_{2n} \end{bmatrix}.$$

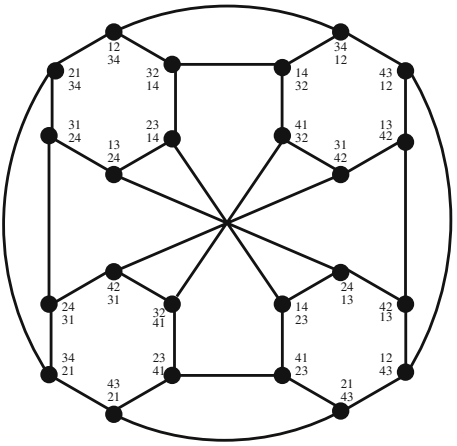


Fig. 1. Matrix-Star graph $MTS_{2,2}$

(2) symbols in the first row are exchanged with symbols in the second row

$$\mathcal{E}(u) = \begin{bmatrix} x_{n+1} & x_{n+2} & \dots & x_j & \dots & x_{2n} \\ x_1 & x_2 & & \dots x_i & \dots & x_n \end{bmatrix}.$$

(3) the first symbol in the first column is exchanged with the first symbol in the second row

$$\mathcal{R}(u) = \begin{bmatrix} x_{n+1} & x_2 & & \dots x_i & \dots & x_n \\ x_1 & x_{n+2} & \dots & x_j & \dots & x_{2n} \end{bmatrix}.$$

Edges in a matrix-star graph can be distinguished as $\mathcal{C}$ *edge* if it is obtained by the operation (1), $\mathcal{E}$ *edge* by (2), and $\mathcal{R}$ *edge* by (3), respectively. From the above definition, $MTS_{2,n}$ consists of $(2n)!$ nodes because it can generate matrices from the permutations of $2n$ symbols, and it is a regular graph of degree $n+1 (n \geq 2)$. If n is 1, then it is a complete graph with 2 nodes because the operation (2) and (3) are identical, and it results in degree of 1. Fig. 1 shows $MTS_{2,2}$ with nodes represented by 2×2 matrices. Throughout the paper, we use terms node and matrix interchangeably.

2.1 Scalability

An interconnection network is said to be *scalable* if it can be easily expanded to a lager network from a smaller size of network. We show scalability of a matrix-star graph $MTS_{2,n}$, $n \geq 2$, to be constructed it from lower-dimensional $MTS_{2,n-1}$. Let $MTS_n\binom{x_n}{x_{2n}}$ be a subgraph of $MTS_{2,n}$ that nth symbol of the first row and $(2n)$th symbol of the second row are fixed as x_n and x_{2n}, respectively. The number of nodes of $MTS_n\binom{x_n}{x_{2n}}$ is $(2n-2)!$ and nodes are connected by $\mathcal{C}_i$, $2 \leq i \leq n-1$, edges and $\mathcal{R}$ edges. Since nodes connected by $\mathcal{E}$ edges in $MTS_n\binom{x_n}{x_{2n}}$ have symbols $\binom{x_{2n}}{x_n}$ in nth column, nodes in $MTS_n\binom{x_n}{x_{2n}}$ can not be connected

by $\mathcal{E}$ edges. If nodes are connected by $\mathcal{E}$ edges, they belong to $MTS_n\binom{x_{2n}}{x_n}$. Let $MTS_n\binom{x_n}{x_{2n}}^*$ be a subgraph of $MTS_{2,n}$ consisting of nodes whose nth column is fixed by $\binom{x_n}{x_{2n}}$ or $\binom{x_{2n}}{x_n}$. For simplicity, we write $MTS_n\binom{a}{b}^*$ as $MTS\binom{a}{b}^*$. Then $MTS\binom{x_n}{x_{2n}}^*$ is a graph consisting of $MTS_n\binom{x_n}{x_{2n}}$ and $MTS_n\binom{x_{2n}}{x_n}$ in which nodes are connected by $\mathcal{E}$ edges. Since the number of nodes of $MTS\binom{x_n}{x_{2n}}^*$ is $2(2n-2)!$ and nodes are connected by $\mathcal{C}_i$, $2 \le i \le n-1$, $\mathcal{R}$, and $\mathcal{E}$ edges, $MTS_{2,n}$ consists of $n(2n-1)$ $MTS\binom{x_n}{x_{2n}}^*$, and each $MTS\binom{x_n}{x_{2n}}^*$ are connected to $MTS\binom{x_i}{x_n}^*$ and $MTS\binom{x_i}{x_{2n}}^*$, $1 \le x_i \le 2n$, by $\mathcal{C}_n$ edges.

Now, we show how to construct $MTS\binom{a}{b}^*$, $1 \le a, b \le 2n$, from $MTS_{2,n-1}$, and then construct $MTS_{2,n}$ by connecting them. Let $MTS'_{2,n-1}$ and $MTS''_{2,n-1}$ be graphs constructed by adding symbols x_n and x_{2n} in nth column to nodes of $MTS_{2,n-1}$ such that nodes of $MTS'_{2,n-1}$ are represented by $2 \times n$ matrix $\begin{bmatrix} * & * & * & x_n \\ * & * & * & x_{2n} \end{bmatrix}$ and nodes of $MTS''_{2,n-1}$ are represented by $2 \times n$ matrix $\begin{bmatrix} * & * & * & x_{2n} \\ * & * & * & x_n \end{bmatrix}$. Each node in $MTS'_{2,n-1}$ connected by $\mathcal{C}_i$, $2 \le i \le n-1$, and $\mathcal{R}$ edges are connected to nodes in $MTS'_{2,n-1}$, but each node connected by $\mathcal{E}$ edges can not be connected to a node in it. Similarly, consider for nodes in $MTS''_{2,n-1}$. That is, nodes connected by $\mathcal{E}$ edges to nodes in $MTS'_{2,n-1}$ belong to $MTS''_{2,n-1}$, and nodes connected by $\mathcal{E}$ edges to nodes in $MTS''_{2,n-1}$ belong to $MTS'_{2,n-1}$. In addition, $MTS'_{2,n-1}$ is isomorphic to $MTS_n\binom{x_n}{x_{2n}}$ and $MTS''_{2,n-1}$ is isomorphic to $MTS_n\binom{x_{2n}}{x_n}$. The graph connecting nodes of $MTS'_{2,n-1}$ and $MTS''_{2,n-1}$ by $\mathcal{E}$ edges is isomorphic to $MTS\binom{x_n}{x_{2n}}^*$. Thus, the number of subgraphs of $MTS\binom{a}{b}^*$ is $n(2n-1)$, and connecting them by $\mathcal{C}_n$ edges results in $MTS_{2,n}$. It shows that the matrix-star graph $MTS_{2,n}$ is scalable. The structure of a matrix-star graph $MTS_{2,n}$ that is divided into $n(2n-1)$ subgraphs $MTS\binom{a}{b}^*$ is

$$\begin{array}{ccccc} MTS\binom{1}{2}^* & MTS\binom{1}{3}^* & \cdots & MTS\binom{1}{2n-1}^* & MTS\binom{1}{2n}^* \\ & MTS\binom{2}{3}^* & \cdots & MTS\binom{2}{2n-1}^* & MTS\binom{2}{2n}^* \\ & & \vdots & \vdots & \vdots \\ & & & MTS\binom{2n-2}{2n-1}^* & MTS\binom{2n-2}{2n}^* \\ & & & & MTS\binom{2n-1}{2n}^* \end{array}$$

2.2 Connectivity

A graph G is said to have the connectivity of k if G is divided into subgraphs or trivial graphs by removing at most k nodes. When G has the connectivity of its degree, it is said to be maximally fault tolerant[1]. Similarly, we can define edge connectivity. In interconnection networks, node connectivity or edge connectivity is an important measurement to evaluate a network as still functional, which means non-faulty nodes in the network remain as connected. Let node connectivity of G be $\kappa(G)$, edge connectivity be $\lambda(G)$, and degree be $\delta(G)$. Then, it has been known that $\kappa(G) \le \lambda(G) \le \delta(G)$[1]. We will show a matrix-star graph $MTS_{2,n}$ is maximally fault tolerant.

Lemma 1. *$\kappa(MTS_{2,n}) = n+1$, where $n \geq 2$.*

Proof. To show the node connectivity of $MTS_{2,n}$ is $n+1$, we show that the resulting graph, after removing n nodes from $MTS_{2,n}$, is still connected. Let X be a set of n nodes that will be removed from $MTS_{2,n}$. Then, we discuss the connectivity of the resulting graph $MTS_{2,n} - X$ in two cases based on the location of nodes in X.

Case 1. nodes in X are located in a $MTS\binom{a}{b}^*$, $1 \leq a, b \leq 2n$.

$MTS\binom{a}{b}^*$ is a subgraph of $MTS_{2,n}$ whose nth column is fixed as $\binom{a}{b}$ or $\binom{b}{a}$, and nodes are connected by $\mathcal{C}_i$, $2 \leq i \leq n-1$, $\mathcal{R}$, or $\mathcal{E}$ edges. Thus degree of nodes in $MTS\binom{a}{b}^*$ is n. For a node u of $MTS\binom{a}{b}^*$, if n nodes incident on u are identical to X, $MTS\binom{a}{b}^*$ is divided into $MTS\binom{a}{b}^* - X$ and a trivial graph u. However, each node of $MTS\binom{a}{b}^*$ is connected to a node in $MTS\binom{x}{a}^*$ or $MTS\binom{x}{b}^*$, $1 \leq x \leq 2n$ by $\mathcal{C}_n$ edges, and all other non-faulty nodes are connected each other, Thus, when nodes in X are located in a subgraph $MTS\binom{a}{b}^*$ of $MTS_{2,n}$, $MTS_{2,n} - X$ is connected.

Case 2. nodes in X are located in more than two subgraphs $MTS\binom{a}{b}^*$, $1 \leq a, b \leq 2n$.

Since nodes in X are located in more than two subgraphs $MTS\binom{a}{b}^*$, the maximum number of nodes that can be removed in a subgraph is at most $n-1$. Further, since degree of nodes in a subgraph is n, non-faulty nodes of any subgraph are connected. Thus, it is easy to see that $MTS_{2,n} - X$ is connected.

From the above discussion, $MTS_{2,n}$ is connected even if at most n nodes are removed from it. It shows that $\kappa(MTS_{2,n}) \geq n+1$. Also, $\kappa(MTS_{2,n}) \leq n+1$ because $MTS_{2,n}$ is a regular graph of degree $n+1$. Thus $\kappa(MTS_{2,n}) = n+1$. □

3 Routing Algorithm and Diameter

In this section, we present a routing algorithm and derive the diameter of a matrix-star graph $MTS_{2,n}$. Let a node S be the source node and a node T be the destination node in the matrix-star graph $MTS_{2,n}$, then the routing path to send a message from S to T can be regarded as the process of changing the symbols of S to those of T. Algorithm 1 shows a routing algorithm to send a message from S to T in $MTS_{2,n}$.

The algorithm **Routing** applies matrix operations $\mathcal{C}$, $\mathcal{E}$ or $\mathcal{R}$ to a source node repeatedly until its matrix representation is the same as that of a destination node. For example, for a source node $S = \begin{bmatrix} 2 & 6 & 3 \\ 4 & 5 & 1 \end{bmatrix}$ and a destination node $T = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$, the algorithm **Routing** finds a routing path as follows: $\begin{bmatrix} 2 & 6 & 3 \\ 4 & 5 & 1 \end{bmatrix} \xrightarrow{\mathcal{C}_2} \begin{bmatrix} 6 & 2 & 3 \\ 4 & 5 & 1 \end{bmatrix} \xrightarrow{\mathcal{E}} \begin{bmatrix} 4 & 5 & 1 \\ 6 & 2 & 3 \end{bmatrix} \xrightarrow{\mathcal{C}_3} \begin{bmatrix} 1 & 5 & 4 \\ 6 & 2 & 3 \end{bmatrix} \xrightarrow{\mathcal{R}} \begin{bmatrix} 6 & 5 & 4 \\ 1 & 2 & 3 \end{bmatrix} \xrightarrow{\mathcal{C}_3} \begin{bmatrix} 4 & 5 & 6 \\ 1 & 2 & 3 \end{bmatrix} \xrightarrow{\mathcal{E}} \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$.

Algorithm 1. Routing algorithm in a matrix-star graph

S_i : symbols in ith row of a node S
$S_{i,j}$: a symbol in ith row and jth column of a node S
$\{S_i\}$: a set of symbols in S_i
$swap(A, B)$: exchange A and B
$\mathcal{M}(S)$: apply a matrix operation $\mathcal{M}$ to S, where $\mathcal{M}$ is $\mathcal{C}$, $\mathcal{E}$, and $\mathcal{R}$
Let S be a source node and T be a destination node in $MTS_{2,n}$.
Initially, $A = 1$ and $B = 2$.
begin
  if $|\{S_1\} \cap \{T_1\}| < n/2$ **then**
    $swap(A, B)$
    while $|\{S_1\} \cap \{T_A\}| \neq n$ **do**
      for each $i(i \geq 2)$ **do**
        if $(S_{1,1} = T_{A,i})$ or $(S_{1,1} = T_{A,1}$ and $S_{1,i} \in T_B)$ **then**
          $S = \mathcal{C}_i(S)$
        if $S_{1,1} \in T_B$ **then**
          if $S_{2,1} \in T_A$ **then**
            $S = \mathcal{R}(S)$
          else
            $S = \mathcal{E}(S)$; $swap(A, B)$
    while $S_1 \neq T_A$ or $S_2 \neq T_B$ **do**
      for each $i(i \geq 2)$ **do**
        if $(S_{1,1} = T_{A,i})$ or $(S_{1,1} = T_{A,1}$ and $S_{1,i} \neq T_{A,i})$ **then**
          $S = \mathcal{C}_i(S)$
        else
          $S = \mathcal{E}(S)$; $swap(A, B)$
    if $S \neq T$ **then**
      $S = \mathcal{E}(S)$
end

Table 1. Network cost of the star graph and its variations

Network Model	Size	Degree	Diameter	Network Cost
Star(S_{2n})	$(2n)!$	$2n-1$	$\lfloor 3n - \frac{3}{2} \rfloor$	$\approx 6n^2$
Bubblesort Star(BS_{2n})	$(2n)!$	$4n-3$	$\lfloor 3n - \frac{3}{2} \rfloor$	$\approx 12n^2$
Transposition(T_{2n})	$(2n)!$	$n(2n-1)$	$2n-1$	$\approx 4n^3$
Macro-Star($MS(2,n)$)	$(2n+1)!$	$n+1$	$5n+2.5$	$\approx 5n^2$
Matrix-Star($MTS_{2,n}$)	$(2n)!$	$n+1$	$3.5n+2$	$\approx 3.5n^2$

Diameter is the maximum number of communication links when a message is forwarded from a node to the other node via a shortest routing path. The diameter of $MTS_{2,n}$ is given in the following theorem, and comparisons between $MTS_{2,n}$ and other interconnection networks are presented in Table 1.

Theorem 1. *The diameter of a matrix-star graph* $MTS_{2,n}$ *is bounded by* $3.5n+2$.

4 Broadcasting Algorithm

Let $MTS_n\binom{*}{x_1}$ be a subgraph of $MTS_{2,n}$ consisting of nodes whose nth symbol of the second row is fixed as x_1, $MTS_n\binom{x_2}{x_1}$ be a subgraph consisting of nodes whose symbols in n column are fixed as x_2 and x_1, $MTS_{n-1}\begin{pmatrix} * & x_2 \\ x_3 & x_1 \end{pmatrix}$ be a subgraph of $MTS_n\binom{x_2}{x_1}$ consisting of nodes whose $(n-1)$th symbol of the second row is x_3, and so on. In Fig. 1, $\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$ is a node of $MTS_2\binom{*}{4}$ and $MTS_1\begin{pmatrix} * & 2 \\ 3 & 4 \end{pmatrix}$ which is a subgraph of $MTS_2\binom{2}{4}$.

Definition 1. *For a node u of $MTS_{2,n}$, we define the sequence $\langle s_1, s_2, \ldots, s_k\rangle$ to represent an edge sequence applied to the node u such that u sends message to u' via s_1 edge at first time step, and then u and u' send message to u'', u''' via s_2 edges, and so on, where s_i is $\mathcal{C}$, $\mathcal{R}$, or $\mathcal{E}$.*

Lemma 2. *For a node $u = \begin{bmatrix} x_1 & x_2 & \cdots & x_i & x_n \\ x_{n+1} & x_{n+2} & \cdots & x_j & x_{2n} \end{bmatrix}$, if an edge sequence $\langle \mathcal{C}_2, \ldots, \mathcal{C}_n, \mathcal{E}\rangle$ is applied to u, then at least one node in each subgraph $MTS_n\binom{*}{x_i}$, $1 \le i \le n$, receives a message originated at u.*

Proof. A node u is in $MTS_n\binom{*}{x_{2n}}$. Within $n-2$ steps after an edge sequence $\langle \mathcal{C}_2, \ldots, \mathcal{C}_{n-1}\rangle$ is applied to u, each node $\begin{bmatrix} x_i & x_2 & \cdots & x_1 & x_n \\ x_{n+1} & x_{n+2} & \cdots & x_j & x_{2n} \end{bmatrix}$, $2 \le i \le n$, incident on u can receive the message. At the $n-1$ step, if a message is sent via $\mathcal{C}_n$ edges, nodes including $\begin{bmatrix} x_n & x_2 & \cdots & x_i & x_1 \\ x_{n+1} & x_{n+2} & \cdots & x_j & x_{2n} \end{bmatrix}$ and $\begin{bmatrix} x_n & x_2 & \cdots & x_1 & x_i \\ x_{n+1} & x_{n+2} & \cdots & x_j & x_{2n} \end{bmatrix}$, contain the message from u. Now, if a message is sent via $\mathcal{E}$ edges, then at least one node whose symbols of nth column is $\binom{x_{2n}}{x_i}$, $1 \le i \le n$, can receive the message. Thus, at the n steps after the edge sequence $\langle \mathcal{C}_2, \ldots, \mathcal{C}_n, \mathcal{E}\rangle$ is applied to u, at least one node in each subgraph $MTS_n\binom{*}{x_i}$, $1 \le i \le n$, receives the message originated at u. □

The following Lemmas come directly from Lemma 2.

Lemma 3. *For a node $u = \begin{bmatrix} x_1 & x_2 & \cdots & x_i & \cdots & x_n \\ x_{n+1} & x_{n+2} & \cdots & x_j & \cdots & x_{2n} \end{bmatrix}$, if an edge sequence $\langle \mathcal{E}, \mathcal{C}_2, \ldots, \mathcal{C}_n, \mathcal{E}\rangle$ is applied to u, then at least one node in each subgraph $MTS_n\binom{*}{x_j}$, $n+1 \le j \le 2n$, receives a message originated at u.*

Lemma 4. *For a node u in $MTS_n\binom{*}{x_{2n}}$, if an edge sequence $\langle \mathcal{C}_2, \ldots, \mathcal{C}_n\rangle$ is applied to u, then at least one node in each subgraph $MTS_n\binom{x_j}{x_{2n}}$, $1 \le j(\neq i) \le n$, receives a message originated at u.*

Lemma 5. *For a node u in $MTS_n\binom{*}{x_{2n}}$, if an edge sequence $\langle \mathcal{E}, \mathcal{C}_2, \ldots, \mathcal{C}_{n-1}, \mathcal{R}, \mathcal{E}, \mathcal{C}_n\rangle$ is applied to u, then one node in each subgraph $MTS_n\binom{x_j}{x_{2n}}$, $n+1 \le j(\neq i) < 2n$, receives a message originated at u.*

Now, we present an algorithm called Broadcasting in Algorithm 2.

Algorithm 2. Broadcasting algorithm in a matrix-star graph

Let u be a source node and M be a message of u.
begin
step 1. for $i = n$ **downto** 2 **do**
1.1 broadcast M to nodes in each subgraph $MTS_i \begin{pmatrix} * & \cdots \\ x & \cdots \end{pmatrix}$, $1 \le x \le 2n$, of the graphs whose symbols from $(i+1)$th column to nth column are fixed in the previous iterations
1.2 broadcast M to nodes in each subgraph $MTS_i \begin{pmatrix} x' & \cdots \\ x & \cdots \end{pmatrix}$, $1 \le x'(\neq x) \le 2n$, of the graphs whose symbols from $(i+1)$th column to nth column are fixed in the previous iterations
step 2. broadcast M via $\mathcal{R}$ edge in each subgraph $MTS_2 \begin{pmatrix} x' & \cdots \\ x & \cdots \end{pmatrix}$, $x' \neq x$
end

Theorem 2. *For a source node u, in time $O(n^2)$, the algorithm* Broadcasting *sends messages to all other nodes.*

Proof. We first show that for a source node u and its message M, all other nodes receive M after performing the algorithm **Broadcasting**. A matrix-star graph $MTS_{2,n}$ can be divided into $2n$ subgraph of $MTS_n\binom{*}{x}$, $1 \le x \le 2n$, and each subgraph can be further divided into $2n-1$ subgraphs of $MTS_n\binom{x'}{x}$, $1 \le x'(\neq x) \le 2n$. By Lemma 2 and Lemma 3, we can verify that at least one node in each $MTS_n\binom{*}{x}$, $1 \le x \le 2n$, can receive M. Also, by Lemma 4 and Lemma 5, at least one node in each $MTS_n\binom{x'}{x}$, $1 \le x'(\neq x) \le 2n$, can receive M. Thus, at the first iteration of step 1, at least $2n(2n-1)$ nodes can receive the message M. This process continues until n is 2, and in each iteration of step 1, two symbols are fixed. Therefore, after step 1, at least one node in each subgraph $MTS_2 \begin{pmatrix} * & x''' & \cdots & x' \\ * & x'' & \cdots & x \end{pmatrix}$ receives the message M and the number of nodes having the message M is at least $2n(2n-1)(2n-2)\cdots 4 \cdot 3$. Finally, M is sent via $\mathcal{R}$ edges in each subgraph $MTS_2 \begin{pmatrix} * & x''' & \cdots & x' \\ * & x'' & \cdots & x \end{pmatrix}$ in step 2. It shows that after performing the algorithm **Broadcasting**, all $(2n)!$ nodes in $MTS_{2,n}$ have the message M.

Next, we discuss about time complexity of the algorithm **Broadcasting**. Since the algorithm broadcasting performs broadcasting concurrently in each subgraph, time complexity can be analyzed by the length of an edge sequence applied in each step. From Lemma 2 and Lemma 3, step 1.1 can be done by applying edge sequences $\langle \mathcal{C}_2, \ldots, \mathcal{C}_n, \mathcal{E} \rangle$ and $\langle \mathcal{E}, \mathcal{C}_2, \ldots, \mathcal{C}_n, \mathcal{E} \rangle$ to nodes having M in subgraphs. Consider the first iteration. After sending M via $\mathcal{E}$ edge from a node u, the former edge sequence $\langle \mathcal{C}_2, \ldots, \mathcal{C}_n, \mathcal{E} \rangle$ can be applied to u in the latter edge sequence. Thus, the length of the edge sequence applied in step 1.1 is at most $n+1$. Also, from Lemma 4 and Lemma 5, step 1.1 can be done by applying edge sequences $\langle \mathcal{C}_2, \ldots, \mathcal{C}_n \rangle$ and $\langle \mathcal{E}, \mathcal{C}_2, \ldots, \mathcal{C}_{n-1}, \mathcal{R}, \mathcal{E}, \mathcal{C}_n \rangle$ to nodes having M at step 1.1. Similarly, the former edge sequence $\langle \mathcal{C}_2, \ldots, \mathcal{C}_n \rangle$ can be contained in the letter edge sequence. Thus, the length of edge sequence applied in step 1.2 is at

most $n+2$. Since step 1.1 and step 1.2 executes recursively, the total length of edge sequence of step 1 is at most n^2+4n-5. Step 2 takes one time step in each subgraph MTS_2. Therefore, the running time of the algorithm **Broadcasting** is bounded by $O(n^2)$. □

5 Further Generalization of Matrix-Star Graphs

A $2 \times n$ matrix-star graph $MTS_{2,n}$ can be further generalized to a $k \times n$ matrix star graph $MTS_{k,n}$, $k \geq 2$ and a $k \times n \times p$ matrix-star graph $MTS_{k,n,p}$. Since $MTS_{k,n}$ is isomorphic to $MTS_{k,n,1}$, we only define $MTS_{k,n,p}$. A matrix-star graph, where a node is represented by a $k \times n \times p$ matrix consisting of knp symbols $\{1, 2, \ldots, knp\}$, and two nodes are connected if and only if a node is obtained by the matrix operation $\mathcal{C}_i$, $2 \leq i \leq n$, $\mathcal{E}_j$, $\mathcal{R}_j$, $1 \leq j \leq k$, and $\mathcal{P}_l$, $\mathcal{D}_l$, $2 \leq 2 \leq p$ as follows: for a node u in $MTS_{k,n,p}$

1. $\mathcal{C}_i(u)$: the first symbol in the first row of the first plane is exchanged with the ith symbol in the first row of the first plan
2. $\mathcal{E}_j(u)$: symbols in the first row of the first plane are exchanged with symbols in the jth row
3. $\mathcal{R}_j(u)$: the first symbol in the first column of the first plane is exchanged with the jth symbol in the first column of the first plane
4. $\mathcal{P}_l(u)$: the first symbol in the first row of the first plane is exchanged with the first symbol in the first row of the lth plane
5. $\mathcal{D}_l(u)$: symbols in the first plane are exchanged with symbols in the lth plane

From the above definition, a matrix-star graph $MTS_{k,n,p}$ consists of $(knp)!$ symbols, and it is a regular graph of degree $2k+n+2p-5$. Also, matrix-star graph $MTS_{1,1,2}$, $MTS_{1,2,1}$, and $MTS_{2,1,1}$ are isomorphic to a complete graph K_2. The diameter of $MTS_{k,n,p}$ is given in the following theorem, and comparisons between $MTS_{k,k,k}$ and other interconnection networks are presented in Table 2.

Theorem 3. *The diameter of a matrix-star graph $MTS_{k,n,p}$ is bounded by $5knp + 4kn + 1.5p - 1.5$.*

Table 2. Network cost of a macro-star graph and a matrix-star graph

	Star graph S_{n^2}	Macro-Star graph $MS(n-1, n-1)$	Matrix-Star graph $MTS_{k,k,k}$, $k = \sqrt[3]{n^2}$
size	$n^2!$	$(n^2-2n+1)!$	$n^2!$
degree	n^2-1	$2n-3$	$5(\sqrt[3]{n^2}-1)$
diameter	$\lfloor 1.5(n^2-1) \rfloor$	$2.5(n^2-n-1)$	$5n^2+4\sqrt[3]{n^4}$
network cost	$O(n^4)$	$O(n^3)$	$O(n^{2.7})$

6 Conclusions

A matrix-star graph $MTS_{2,n}$ as a class of lower communication cost network was introduced to improve the network cost, defined by degree $\times$ diameter, of the macro-star graph. Its topological properties and communication schemes were discussed to demonstrate its superiority. We also generalized $MTS_{2,n}$ to $MTS_{k,n}$ and $MTS_{k,n,p}$ to further improve the network cost. Specifically, the network cost of a macro-star graph $MS(n-1, n-1)$ with $((n-1)^2+1)!$ nodes, is $O(n^3)$, and the network cost of a matrix-star graph $MTS_{k,k,k}$, $k = \sqrt[3]{n^2}$.

References

1. S. B. Akers, D. Harel, and B. Krishnamurthy: The Star Graph: an attractive alternative to the n-cube, Proc. Int. Conf. on Parallel Processing (1987) 393–400.
2. L. N. Bhuyan and D. P. Agrawal: Generalized hypercube and hyperbus structures for a computer network, IEEE Trans. Comput. **c-33** (1984) 323–333.
3. J. A. Bondy and U. S. R. Murty: Graph theory with applications, 5th printing, American Elsevier Publishing Co., Inc., 1982.
4. Z.-T. Chou, C.-C. Hsu, and J.-P. Chen: Bubblesort star graphs:a new interconnection network, Proc. Int. Parallel Processing Symposium (1996) 41–48.
5. S. Latifi, M. D. Azevedo, and N. Bagherzadeh: The star connected cycle: A fixed-degree network for parallel processing, Proc. Int. Conf. on Parallel Processing (1993) 91–95.
6. S. Latifi, and P. K. Srimani: Transposition networks as a class of fault-tolerant robust network, IEEE Trans. Comput. **45** (1996) 230–238.
7. F. T. Leighton: Introduction to parallel algorithms and architectures: arrays, trees, hypercubes, Morgan Kaufmann Publishers, 1992.
8. I. Stojmenovic: Honeycomb networks: topological properties and communication algorithms, IEEE Tran. Parallel and Distributed Systems **8** (1997) 1036–1042.
9. C.-H. Yeh and E. A. Varvarigos: Macro-star networks: efficient low-degree alternatives to star graphs, IEEE Tran. Parallel and Distributed Systems **9** (1998) 987–1003.

The Channel Assignment Algorithm on RP(k) Networks*

Fang'ai Liu, Xinhua Wang, and Liancheng Xu

ShanDong Normal University,
Computer Science Department, Jinan, China
lfa9125@beelink.com

Abstract. Embedding and channel assignment is a key topic in optical interconnection networks. Based on RP(k) network, a scheme to embed a hypercube into RP(k) is given and the wavelength assignment of realizing the Hypercube communication on RP(k) network is discussed in this paper. By introducing the reverse order of Hypercube, an algorithm to embed the n-Dimension Hypercube into RP(k) is designed, which multiplexes at most $\max\{2, \lfloor 5*N/96 \rfloor\}$ wavelengths. An algorithm to embed the n-Dimension Hypercube into the ring network is also proposed, with its congestion equal to $\lfloor N/3+N/12 \rfloor$. It is a better improvement than the known result, which is equal to $\lfloor N/3+N/4 \rfloor$. The analyses prove that it is easier to realize the Hypercube communication on RP(k) network.

1 Introduction

Interconnection Network is one of key factors for parallel computers. So much research has been done on this aspect. As the advancement of communication technology, all-optical interconnection networks[1,2], whose advantages have been well demonstrated on wide and local area networks, are considered as promising means to increase the performance of interconnection networks for future parallel computers. But there exists many theoretic questions to be solved before all-optical interconnection networks can be used. Progress in wavelength division multiplexing (WDM [1]) technologies have attracted many researcher's attentions and made optical communication a promising choice to meet the increasing demands for higher channel bandwidth and lower communication latency. With the development of optical interconnection technology, super computers with optical interconnection are entering into consideration and are considered as an important means to increase its performances. By WDM technology, on one single fiber, more than 100 channels can be used. So how to make full use of these channels efficiently has been a hot research field in parallel process. Further more, there exists many different communication patterns on the interconnection networks, analyzing the wavelength requirement of these communication patterns on different optical interconnection networks is a basic way to have an insight into the capacity of interconnection networks. For example, on a special optical network, by analyzing the maximum number of wavelength we need to realize hypercube communication, we can evaluate the efficiency of this

* Supported by the National Natural Science Foundation of China under Grant No. 60373063.

T. Srikanthan et al. (Eds.): ACSAC 2005, LNCS 3740, pp. 488 – 498, 2005.

interconnection network. Many researchers have done much work on this field. In [2], the permutation embedding and scheduling in the optical mesh networks is studied and the capacity of the optical mesh is evaluated. In [3], routing and channel assignment for Hypercube communication on optical ring is discussed. In [4,5], the problem of communication performance caused by the optical connections is studied.

In this paper, we discuss the embedding and wavelength assignment of realizing Hypercube communication with $N=2^n$ nodes on the optical RP(k) networks, which is proposed in[6]. An algorithm to embed the n-Dimension Hypercube into RP(k) network is designed, which multiplexes at most $\max\{2, \lfloor 5*2^{n-5}/3 \rfloor\}$ wavelengths. We also propose a new algorithm to embed the n-Dimension Hypercube into the ring, which requires $\lfloor N/3+N/12 \rfloor$ wavelengths at most. This result has improved the result of $\lfloor N/3+N/4 \rfloor$ proposed in [3] greatly.

2 Preliminaries

In this Section, we first give some basic concepts which includes optical WDM network, wavelength assignment and RP(k) network.

2.1 Optical WDM Networks

Optical WDM networks [1] are widely regarded as one of the best choices for providing the huge bandwidth required by future networks. Wavelength Division Multiplexing (WDM) divides the bandwidth of an optical fiber into multiple wavelength channels, so that multiple users can transmit at distinct wavelength channels through the same fiber concurrently. To efficiently utilize the bandwidth resources and eliminate the high cost and bottleneck caused by optoelectronic conversion and processing at intermediate nodes, the end-to-end lightpaths are usually set up between each pair of source-destination nodes. A connection or a lightpath in a WDM network is simulated as an ordered pair of nodes (x,y) corresponding to that a packet is sent from source x to destination y. There are two approaches for establishing a connection in a network whose links are multiplexed with virtual channels. One is called Path Multiplexing (PM), in which the same wavelength has to be used on each link along a path, and the other is called Link Multiplexing (LM), in which different wavelength may be used on different links along this path. The later needs to place wavelength converters [8,10] on intermediate nodes and the cost will be increased. In this paper, we assume that no wavelength converter facility is available in the network. Thus, a connection must use the same wavelength throughout its path. We call this case that the lightpath satisfies the wavelength-continuity constraint.

2.2 Wavelength Assignment

Given a communication pattern, routing and wavelength assignment (RWA[3]) tries to minimize the number of channels to realize a communication requirement by taking into consideration both routing options and wavelength assignment options. The RWA problem can be described as follows.

Given an optical network and a set of all-optical connections, C={(x,y)| where x is the source node and y is the destination }, the problem is :

(a) to find a route from each source nodes to their respective destinations for each connection.
(b) to assign a wavelength to each route so that the same wavelength is assigned to all the links of a particular route. If two routes pass one same link, the two routes are certainly assigned with different wavelengths.
(c) The goal of RWA is to minimize the number of assigned wavelengths.

Many researchers have studied the RWA problem and some results are given in[2,3]. In this paper, we discuss the RWA problem of Hypercube communication on the RP(k) network.

2.3 RP(k) Network

Based on the Petersen graph [Fig.1], we have constructed a new interconnection network of RP(k) in [6]. The interconnection network, RP(k), consists of *k* slices. Each slice has 10 nodes, which are connected as a Petersen graph. The *k* slices are named slice 0, slice 1, and slice *k*-1. Within a slice, there are 10 nodes. We name these nodes as 0, 1, …and 9. The *k* slices are linked together by 10 rings. Each ring consists of k nodes that have the same number on these *k* slices. Thus there are totally 10 rings, named ring 0, ring 1… and ring 9. The nodes in the RP(k) network are named as follows. The address of a node in the RP(k) network consists of two parts, denoted as (*m,n*), where *m* is the number of slices and *n* is the number of the nodes within slice *m*. Certainly, we have 0≤m≤k-1 and 0≤n≤9.

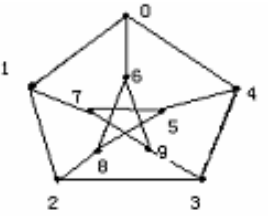


Fig. 1. Petersen Graph

The RP(k) network has many good properties such as small diameter, simple topology and convenience routing schemes[6]. The network has 10*k nodes. Its connectivity degree is 5 and its diameter is $\lfloor k/2 \rfloor + 2$. In addition, the algorithms designed for ring and mesh network can be easily and efficiently embedded into RP(k) network[9]. We have proved[6] that when the number of nodes in the networks is no more than 300, the diameter in the RP(k) network is smaller than that in the 2-D Torus. Especially when the number of nodes in a node group is between 6 and 100, its diameter is approximately half of that in the 2-D Torus. So RP(k) network is more suitable for constructing a parallel computer system with less than 300 nodes. Furthermore if the SMP architecture is adopted on each node, we can link a parallel computer with more than several thousand processors by the RP(k) network.

3 Wavelength Assignment of Hypercube

In order to realize Hypercube communication on RP(k) network, we want to design an algorithm to embed the Hypercube communication pattern into RP(k) network. We hope that the maximum number of connections passing through all the links, which is

called network congestion [9], should be minimized. As we know, the n-Dimension Hypercube consists of $N=2^n$ nodes. For the sake of simplicity, we assume $k=2^{n-3}$. In the following sections, we design an algorithm to embed the Hypercube communication into RP(k) aiming at minimizing its congestion.

3.1 Properties of Hypercube

As we know, the Hypercube has the following property.

Property 1: n-Dimension Hypercube consists of two (n-1)-Dimensional Hypercubes connected by 2^{n-1} connections between them.

The binary identifications of the node pairs, which are linked by these connections, are the same with the last n-1 bits. For example, there exists a link between Node $0XX...XX$ and 1XX...XX ,where X represents 0 or 1.

In addition, any Hypercube with more than 3 dimensions can be regarded as the construction of a number of 3-D Hypercubes according to the connecting regularity. Therefore, our idea is to embed a 3-D Hypercube into a Peterson graph firstly, and then discuss the wavelength assignment of the n-D Hypercube embedded in RP (k) network.

3.2 3-D Hypercube Embedding into the Peterson Graph

Since the 3-D Hypercube can be regarded as a basic element of the n-D Hypercube, we first consider embedding a 3-D Hypercube into a Peterson graph, and then discuss the wavelength assignment of the n-D Hypercube on the RP (k) network.

By enumerating all cases we know that in the view of isomorphism, Peterson graph with two nodes deleted has only two shapes, as can be seen in Fig.2(a) and (b). Here we choose the graph (b) for discussion. We define the mapping from the 3-D Hypercube to Fig.2(b) as follows.

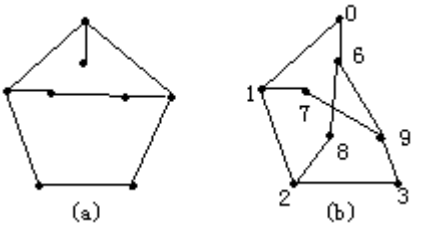


Fig. 2. The Petersen graphs with two nodes deleted

① mapping from the nodes of 3-D Hypercube to the nodes of Fig.2(b):

000---1 011---6 001---0 101---8 010---7 110---9 100---2 111---3

②mapping(by the shortest path) from the edges of 3-D Hypercube to the paths of Fig.2(b) shown in Table 1:

In Table 1, the edges in the bracket are the paths that the mapped edges pass through. The following property can be derived from this mapping method.

Property 2: The congestion is 3 when 3-D Hypercube is embedded in Fig.2(b) by the above mapping method.

Table 1. Mapping from the edges of 3-D Hypercube to the edges of Fig.2(b)

Edges of Hypercube	Paths of Fig.2(b)	Edges of Hypercube	Paths of Fig.2(b)
000—001	1—0	000—010	1—7
000—100	1—2	100—110	2—9(2—3—9)
100—101	2—8	101—001	8—0(8—6—0)
101—111	8—3 (8—2—3)	111—011	3—6(3—9—6)
111—110	3—9	011—010	6—7(6—9—7)
010—110	7—9	011—001	6—0

Therefore, the maximum number of wavelength required to realize the 3-D Hypercube on the Peterson graph is 3. By simple analysis we know that if the embedding is not routed according to the shortest path, the maximum number of wavelength can be reduced to 2.

3.3 n-D Hypercube Embedding into RP(k) Network

From property 1 we know that, n-D Hypercube can be regarded as the connection of two (n-1)-D Hypercubes connected by the 2^{n-1} connections. Thus, the following property can also be derived.

Property 3: n-D Hypercube can be regarded as (n-2)-D Hypercube when the 3-D Hypercube is regarded as the basic element.

As we know, RP(k) network can be regarded as a ring network when the Peterson graph is regarded as the basic element. Therefore, the problem mentioned above can be simplified into the problem of embedding Hypercube into the ring network. This problem of Hypercube embedding into ring has already been studied in [3] and the results are stated in Lemma 1 and 2, where $N=2^n$.

Lemma 1: The number of wavelength required to realize the Hypercube communication on the linear array with N nodes is $\lfloor 2*N/3 \rfloor$.

Lemma 2: The number of wavelength required to realize the Hypercube communication on the ring with N nodes is $\lfloor N/3+N/4 \rfloor$.

From Lemma 2, the following theorem can be derived.

Theorem 1: The number of wavelength required to realize the Hypercube communication with N nodes on RP(k) network is $\lfloor N/24+N/32 \rfloor$, where $N=2^n$ and $k=2^{n-3}$.

Proof: When N is equal to 8,16 and 32, it is easy to know that the number of wavelength required is 2, 2, and 2 respectively. In the case of $N=2^n>16$, from Lemma 2, the number of wavelength required is $\lfloor 2^{n-3}/3+2^{n-3}/4 \rfloor = \lfloor 2^n/24+2^n/32 \rfloor$

considering the Hypercube with 2^{n-3} nodes embedded in the ring according to Lemma 2. Thus the result is proved.

It can be seen from Theorem 1 that realizing the Hypercube communication with 32 nodes in RP(k) network requires only 2 wavelengths and realizing the Hypercube communication with 128 nodes requires only 9 wavelengths.

Property 4: The connections in the n-D Hypercube which connect the two (n-1)-D Hypercubes can be denoted by {(0x,1x)}, where x can be any one of the binary identifications of the node in the (n-1)-D Hypercube.

4 A New Algorithm of Hypercubes Embedding into Rings

In this section, we establish a new mapping method from the Hypercube to ring. Assume that X is an order of a string, let X^{-1} is the reverse order of this sting. For example, if X=a,b,c,d then X^{-1}=d,c,b,a. Obviously, $(X_1X_2)^{-1} = (X_2)^{-1}(X_1)^{-1}$. Now we define the node order of n-D Hypercube X_n recursively as the following:

$$X_1 = 0, 1$$

$$X_2 = 0X_1, (1X_1)^{-1} = 00,01,(10,11)^{-1} = 00,01,11,10$$

.....

$$X_n = 0X_{n-1}, (1X_{n-1})^{-1}$$

In X_n, each node is separated by a comma, for example, the node order of 4-D Hypercube, X_4, can be denoted by

$$X_4 = 0X_3, (1X_3)^{-1} = 0(0X_2, (1X_2)^{-1}), 1(0X_2, (1X_2)^{-1})^{-1}$$

$$= 0000,0001,0011,0010,0110,0111,0101,0100,1100,1101,1111,$$
$$1110,1010,1011,1001,1000$$

X_n is a node sequence of n-D Hypercube, called *reverse order*. Given a ring with 2^n nodes, numbered by 1,2… 2^n. if the *i*th node of X_n is mapped onto node i, where $1 \le i \le 2^n$, then a 1-1 mapping from the nodes of X_n to the nodes of the ring is established. We call this method *reverse mapping*.

Property 5: The first 2^{n-1} nodes of the reverse order X_n form a (n-1)-D Hypercube and the last 2^{n-1} nodes of X_n form another (n-1)-D Hypercube.

Property 6: All the connections, which connect the *i*th node and the $(2^n\text{-}i+1)$th node of X_n (i=1,2,…2^{n-1}), form the all connections between the two (n-1)-D Hypercubes.

Since reverse mapping defines a mapping between the nodes of a Hypercube and a ring. We then define a mapping from a hypercube edges to a path in ring. Let (x,y) is an edge in Hypercube and (x1,y1) are their images of x,y by the reverse mapping. We define the image of edge (x,y) is the shortest path between x1 and y1. These two mappings form an embedding from Hypercube to ring, we then analyze its embedding congestion, that is, the number of wavelength required.

First consider a linear array, not a ring network.

Theorem 2: By the reverse mapping from the n-D Hypercube to the linear array with 2^n nodes, the number of the connections which pass the *i*th edge of the linear array, A(n,i), can be calculated recursively by the following equation.

$$A(n,i)=\begin{cases}2^{n-1} & i=2^{n-1}\\ A(n-1,i)+i & 1\le i<2^{n-1}\\ A(n-1,2^n-i)+2^n-i & 2^{n-1}<i\le 2^n-1\end{cases} \quad (1)$$

Here the number of connections required for the 3-D Hypercube is A(3, 1--7)={3, 4, 5, 4, 5, 4, 3}.

Proof: According to the embedding mapping, we have mapped the nodes of the first (n-1)-D Hypercube of X_n on the first 2^{n-1} nodes of the linear array and the nodes of the second (n-1)-D Hypercube of X_n on the last 2^{n-1} nodes. Thus, the connections between these two Hypercubes all pass through the 2^{n-1}th edge of the linear array. So the congestion on the 2^{n-1}th edge is 2^{n-1}.

Since the (n-1)-D Hypercube is embedded on the first 2^{n-1}-1 edges of the linear array according to the reverse order, the number of connections passing through those edges is the sum of the connections occupied by the (n-1)-D Hypercube and the connections between the two (n-1)-D Hypercubes. Assuming that the number of connections in the *i*th edge occupied by the (n-1)-D Hypercube is A(n-1,i), it can be easily proved that the number of connections between the two (n-1)-D Hypercube passing through the *i*th edge is i. Thus, the total number of connections passing through the *i*th edge is A(n-1,i)+i. Similarly, the number of connections passing through the last 2^{n-1}-1 edges of the linear array can also be calculated according to Formula (1).

Corollary 1: Given n, let MaxA(n) be the maximum obtained from formula (1) and MaxA(n) can be achieved on the edge of Index(n), then Index(n) and MaxA(n) can be calculated by the following formula.

$$Index(n)=\begin{cases}1 & n=2\\ 2*\text{Index}(n-1)+1=(2^n+1)/3 & n \textit{ is} \text{ odd}\\ 2*\text{Index}(n-1)-1=(2^n-1)/3 & n \text{ is even}\end{cases}$$

$$MaxA(n)=\begin{cases}2 & n=2\\ 2*\text{MaxA}(n-1)=2*(2^n-1)/3 & n \textit{ is} \text{ even}\\ 2*\text{MaxA}(n-1)+1=2*(2^n-0.5)/3 & n \textit{ is} \text{ odd}\end{cases}$$

Proof: We prove this result by the mathematical induction considering the dimensions of the Hypercube.

① If n=2, 3, it can be easily known that Index(2)=1 and Index(3)=3=2*Index(2)+1=$(2^3+1)/3$. The result holds in these cases.

② Assuming that the result holds when the dimensions are equal to n-1 or less than n-1, then we discuss the case of n dimensions when n is odd and even respectively.

When n is even, we prove that MaxA(n) reachs the maximum on the edge of 2*Index(n-1)-1 and the maximum is 2*MaxA(n-1)=$2*(2^n-1)/3$.

From the assumption, the result holds when the dimension is less than n. By embedding the first (n-1)-Dimensional Hypercube on the first part of the linear array, the maximum is achieved on the edge of Index(n-1). Therefore,

Index(n-1)=2*Index(n-2)+1=2*(2*Index(n-3)-1)+1=2^2*Index(n-3)-1
=2^2*(2^2*Index(n-5)-1)-1=…=(2^{n-1}+1)/3.

Similarly, it can be calculated that MaxA(n-1)= 2*(2^{n-1}-0.5)/3.

In fact, we have A(n,i)=A(n, 2^n-i). Since the maximum of MaxA(n-1) is achieved on the edge of (2^{n-1}+1)/3, on the edge of 2^n-(2^{n-1}+1)/3=(2^n-1)/3, MaxA(n-1) also obtains the maximum. As 2*Index(n-1)-1=2*(2^{n-1}+1)/3-1=(2^n-1)/3, thus MaxA(n-1) achieves its maximum on the edge of 2*Index(n-1)-1. In the following, we prove that MaxA(n) achieves its maximum on the edge of (2^n-1)/3.

Because of the symmetry characteristics of X_n, we only need to consider the edges from (2^n-1)/3+1 to 2^{n-1}-1. In fact, if 1≤i≤(2^n-2)/3, then A(n, (2^n-1)/3) ≥A(n, (2^n-1)/3+i). Although the number of connections on the edge of (2^n-1)/3+i increases by i because of the connections between the two (n-1)-D Hypercubes, it also decreases by at least i because of the connections of the Hypercube whose dimension is less than n-1. Therefore, A(n, (2^n-1)/3) is the maximum.

As can be known, MaxA(n) is the sum of two parts: MaxA(n-1) and the connections passing through the two (n-1)-D Hypercubes. Since the second part contains (2^n-1)/3 connections, then

MaxA(n)=MaxA(n-1)+(2^n-1)/3=(2^n-1)/3+(2^n-1)/3=2*MaxA(n-1)=2* (2^n-1)/3.

Thus, we prove that the result in Corollary 1 is true when n is even.

When n is odd, it can be proved similarly. Here we ignore the details.

From the assumption, we know that Corollary 1 holds.

Corollary 2: The maximum of MaxA(n) in Theorem 2 can be expressed by $\lfloor 2^{n+1}/3 \rfloor$.

The result obtained from Corollary 2 is identical with that appeared in [3]. In [3], it has been proved that the minimum congestion for the embedding of n-D Hypercube on the linear array is $\lfloor 2^{n+1}/3 \rfloor$. It can be concluded that our algorithm is optimal. In the following, we discuss the embedding of n-D Hypercube in the ring and derive a more optimal result than that appeared in [3].

Theorem 3: An algorithm can be designed to embed the Hypercube communication with N=2^n nodes into the ring network and the number of wavelength required on the *i*th edge of the ring can be calculated by the following equation.

$$C(n,i)=\begin{cases} 2^{n-2} & i = 2^{n-1} \ or \ 2^n \\ A(n-1,i)+2^{n-2}-i & 1 \le i < 2^{n-2} \\ A(n-1,i)+i-2^{n-2} & 2^{n-2} \le i < 2^{n-1} \\ A(n-1,2^n-i)+1.5*2^{n-1}-i & 2^{n-1} < i < 1.5*2^{n-1} \\ A(n-1,2^n-i)+i-1.5*2^{n-1} & 1.5*2^{n-1} \le i < 2^n \end{cases} \quad (2)$$

where A(3, 1--7)={ 3, 4, 5, 4, 5, 4, 3 }.

Proof: Firstly, we embed the nodes of the first (n-1)-D Hypercube on the ring from the node of 1 to 2^{n-1} according the reverse order and embed the nodes of the second

(n-1)-D Hypercube on the ring from the node of $1+2^{n-1}$ to 2^n. Secondly, we consider the connections between the two (n-1)-D Hypercubes. In the reverse order, X_n can be divided into four node sets ($1--2^{n-2}$, $2^{n-2}+1--2^{n-1}$, $2^{n-1}+1--1.5*2^{n-1}$ and $1.5*2^{n-1}+1--2^n$) and each set contains 2^{n-2} nodes. At the same time, the connections between the two (n-1)-D Hypercubes can be regarded as two parts. One part includes connections that connect between the nodes set of $2^{n-2}+1--2^{n-1}$ and the nodes set of $2^{n-1}+1--1.5*2^{n-1}$. Let these connections all pass through the edge of $(2^{n-1},2^{n-1}+1)$ on the ring. The other part includes the connections that connect between the nodes set of $1--2^{n-2}$ and the nodes set of $1.5*2^{n-2}+1--2^n$. We assume that these connections all pass through the edge of $(1,2^n)$ on the ring.

Accordingly, C(n,i) includes two parts. One part is the number of connections in the (n-1)-D Hypercube, which is A(n-1,i). The other part is the number of connections between the two (n-1)-D Hypercubes, which is the number expressed in equation (2).

Then, how to obtain MaxC(n), the maximum of C(n,i)? Similar with Corollary 1, we give Corollary 3.

Corollary 3: The maximum congestion MaxC(n) and the edge Indexc(n) achieving this maximum can be calculated by the following equations.

$$Indexc(n)=\begin{cases}1 & n=4\\ 2*\text{Index}(n-1)+1=(2^{n-2}+1)/3 & n\ is\ odd\\ 2*\text{Index}(n-1)-1=(2^{n-2}-1)/3 & n\ \text{is even}\end{cases}$$

$$MaxC(n)=\begin{cases}6 & n=4\\ 2*\text{MaxC}(n-1)=(5*2^{n-2}-2)/3 & n\ is\ even\\ 2*\text{MaxC}(n-1)+1=(5*2^{n-2}-1)/3 & n\ is\ odd\end{cases}$$

Similar with Theorem 1, Corollary 3 can be proved. Here we ignore the details of the proving for the sake of simplicity. Compared with Lemma 2, the result of Corollary 3 is a better improvement than the known result[3]. MaxC(n) in Corollary 3 can also be expressed by $\lfloor N/3+N/12\rfloor$, while the result in Lemma 2 is $\lfloor N/3+N/4\rfloor$. The comparison of maximum wavelengths by MaxC(n) and Lemma 2 is shown in Table 2.

Table 2. The comparison of maximum wavelengths in MaxC(n) and Lemma 2

Dimension n	4	6	8	10	12	14
MaxC(n)	6	26	106	426	1706	6826
Lemma 2	9	37	149	597	2389	9557

Based on Corollary 3, Theorem 1 can be improved.

Theorem 4: The number of wavelength required to realize the Hypercube communication with N nodes on the optical RP(k) network is $\max\{2, \lfloor 5*2^{n-5}/3\rfloor\}$, where $N=2^n$ and $k=2^{n-3}$.

Some numbers of wavelength required to realize Hypercube communication in the optical RP(k) network are shown in Table 3. For example, realizing the Hypercube

communication with 256 nodes on the optical RP(32) network only requires 13 wavelength. In this case, it is feasible for the current technology to supply more than 10 wavelengths in one fiber.

Table 3. The wavelengths needed to embed Hypercube into RP(k)

HypercubeDimension	6	7	8	9	10	11
Number of Hypercube node	64	128	256	512	1024	2048
RP(k) wavelength	3	6	13	26	53	106

5 Conclusions

Analyzing special communication patterns on interconnection networks is a basic and efficient way to have an insight into the capacity of interconnection networks. In this paper, we discuss the wavelength assignment of realizing Hypercube communication on optical RP(k) networks. By introducing the reverse mapping of the Hypercube, an algorithm to embed the n-D Hypercube into the RP(k) network is designed, which multiplexes at most $\max\{2, \lfloor 5*2^{n-5}/3 \rfloor\}$ wavelengths. two efficient algorithms to embed the n-D Hypercube into the linear array and ring network are also proposed. The algorithm to embed the n-D Hypercube into ring network, which requires $\lfloor 2^n/3+2^n/12 \rfloor$ wavelengths, is a better improvement than the known result, which is $\lfloor 2^n/3+2^n/4 \rfloor$. The analyses prove that it is easier to realize the Hypercube communication on RP(k) network. These results are obtained without considering wavelength conversion at each node. The research when some nodes with wavelength converters is our future direction.

References

1. R.Ramaswani and K.N. Sivarajan, Optical Network: A practical Perspective, Morgan Kanfmann Publishers, San Francisco, USA(2002).
2. Qiao Chunming and Mei Yousong, Off-Line Permutation Embedding and Scheduling in Multiplexed Optical Networks with Regular Topologies, IEEE/ACM Tran. On Networking, Vol.7(2)(1999)241-250
3. Xin Yuan and Melhem R., Optimal Routing and Channel Assignments for Hypercube Communication on Optical Mesh-like Processor Arrays, In: Johnsson S. L., ed., Fifth International Conference on Massively Parallel Processing Using Optical Interconnection, Las Vegas, NV, IEEE Press(1998).
4. Xin Yuan, Rami Melhem and Rajiv Gupta, Distributed Path Reservation Algorithm for Multiplexed All-Optical Interconnection Networks, IEEE Tran. On Computer, Vol.48 (12)(1999)1355-1363.
5. Xin Yuan, Melhem R. and Gupa R., Performance of Multi--hop Communications Using Logical Topologies on Optical Torus Networks, Journal of Parallel and Distributed Computing, Vol 61(6) (2001) 748-766.
6. Liu Fangai, Liu zhiyong and Qiao xiangzhen, A Practical Interconnection Network RP(k) and its Routing Algorithms, Science in China(Series F), Vol. 44 (6)(2001) 461- 473.

7. Shen Xiaojun, Liang weifa, Hu Qing, On Embedding Between 2D Meshes of the Same Size, IEEE Trans. Computer, Vol. 46(8)(1997)880-888.
8. Lu Ruan, Ding Du, Xiaodong Hu, Converter Placement Supporting Broadcast in WDM Optical Networks, IEEE Transaction on Computer,Vol.50(7)(2001):750--758.
9. Liu Fang'ai, Liu Zhiyong and Zhang Yongsheng, The embedding of Rings and Meshes into RP(k) networks, Science in China(Series F),Vol.47(5)(2004)669--680.
10. Xiao-Hua Jia, Ding-Zhu Du, Xiao-Dong Hu, Optimization of Wavelength Assignment for QoS Multicast in WDM Networks, IEEE Transactions on Communications, Vol. 49(2)(2001)341--351.

Extending Address Space of IP Networks with Hierarchical Addressing

Tingrong Lu[1], Chengcheng Sui[1], Yushu Ma[2], Jinsong Zhao[3], and Yongtian Yang[1]

[1] Dept. of Computer Science, Harbin Engineering University, Harbin 150001, China
lutingrong@hrbeu.edu.cn
[2] Dept. of Computer Science, University of Petroleum, Beijing 102200, China
[3] NEC Solutions Asia Pacific Pte Ltd, Singapore 099253

Abstract. To present a solution to the problem of address space exhaustion in the Internet, this paper proposes a multi-tier addressing architecture of the Internet (IPEA), illustrates the address space and addressing scheme of IPEA, defines the format of IP packet with extension address, and hierarchical routing algorithm. This paper analyzes and evaluates the performance of IPEA with simulation, and concludes that IPEA has such advantages: first, it provides a solution to the problem of address space exhaustion in the Internet; second, it reduces the routing table length, helps dealing with routing table explosion; third, it demands little change on the IPv4 addressing scheme, easy for transition; finally it makes the autonomous network more manageable by using private address.

1 Introduction

IPv4 is facing problems of address space exhaustion in the Internet, routing table explosion, management of networks, new demands from the new applications. There are two major approaches in dealing with these problems: to design brand new IP protocol [2], or to make patches on the IPv4. Among the first category, the next generation IP protocols--IPv6 [5] is released. Among the second category, private IP address [19], temporarily assigned IP address [23] and DHCP [6], more efficiently use of IP address--CIDR [10], multiplexing of IP address--NAT [7], are put forward.

Due to the interoperability with IPv4, IPv6 hasn't been deployed widely yet. Those patches on IPv4 cannot solve the problems thoroughly, and NAT produces difficulties for some end to end applications.

Is it possible to deploy multi-layer, variable-length address in IP networks like in the telephony systems? How to number and how long the number is are decided locally, the local number prefixed with area code and country code will be globally unique. This is hierarchical naming and addressing [20, 4]. TCP/IP has a hierarchical model very early, and the IP address is hierarchically divided into network number and host number too.

To expand address space of IP networks, a hierarchical addressing architecture (IP with Extension Addresses, IPEA) is put forward in this paper. The contribution in this paper is that different classes of address are proposed to designate different levels in the hierarchical networks and routers in different levels, and the associated

T. Srikanthan et al. (Eds.): ACSAC 2005, LNCS 3740, pp. 499 – 508, 2005.

hierarchical routing algorithm is analyzed. The differentiation of levels in the network saves the intermediate routers of altering addresses in the packets in the procedure of packets forwarding.

Related work is in section 2, hierarchical addressing architecture is illustrated in section 3, the format of IPEA packet is designed in section 4, associated hierarchical routing algorithm, correctness proof and the complexity analysis are illustrated in section 5, the interoperability between IPEA and IPv4 is illustrated in section 6, performance analysis in section 7, simulation results in section 8, finally, the conclusion in section 9.

2 Related Work

Similar works are PIP [8], IPNL [9], IP4+4 [22], Nimrod [3], and layered naming [1], etc. Francis' PIP brings forward multi-layer, variable-length addressing, address can be reused in different domains. But there are two problems with this scheme: 1) PIP domain has no hierarchy, its address string is an extension of IPv4 source routing mechanism; 2) in the forwarding procedure of its packet, source and destination address has to be reverted by every hop, or the router cannot know which address of the packet should be forwarded based on, which increases the computation overhead of the router. IPNL has a 2-byte realm number as part of address which makes the parts of IPNL address cannot be treated uniformly, and location field in IPNL packet has to be altered by intermediate routers. IP4+4 includes address swapping too. Nimrod, layered naming, etc, deploy a hierarchical naming to address services and data, by establishing some DNS-like mechanism to translate hierarchical name to IP address. This category of schemes focuses on multiplexing names of services and data on IP address, and does not expand IP address space.

Hierarchical routing is originated from McQuillan [14]. Kleinrock and Kamoun [11] analyzed, in McQuillan's scheme, the relation between the reduction of routing table length and the increase of routing path length regarding to the increase of the levels of the networks. They conclude that hierarchical networks can reduce routing table length significantly, while the increase of routing path length is trivial. In [11], each node in the network is in the lowest level clusters, and all super clusters are virtual. The routing table in each node has an entry for each node in the local cluster and each cluster which belongs to the same super-cluster as the node in question does. In IPEA, nodes can be in any cluster in any level. The routing table in each node has an entry only for each node in the local cluster (in each border node has in addition several more entries for border nodes in other clusters). Tsai et al. observed the impact of update period, and some other parameters, on the performance of hierarchical routing [21]. Based on distributed Bellman-Ford algorithm, topology broadcast algorithm, distributed Dijkstra algorithm, many hierarchical routing algorithms have been put forward [18, 12, 15, 13]. In this category of works, nodes in the network are identified using Dewey notation [11], no hierarchical naming or addressing has been explicitly mentioned, but hierarchical routing is proved to be feasible and effective.

3 Hierarchical Address Space and Addressing Scheme in IP Networks

Definition 1. According to the definition of class A, B... address in IPv4, we generalize it to class Z address.

Definition 2. <IPEA (global) address> ::= [<Class A address>][<Class B address>][<Class C address >]...[<Class Z address >][<IPX address> | <NetBIOS name>]; the first address fragment of IPEA (global) address is called top-level address of IPEA (global) address, it must be a valid IPv4 address, these addresses belong to IPv4 routing domain.

To simplify the illustration, the case of that the extension addresses might be IPX address or NetBIOS names is not considered in this paper.

Definition 3. The last fragment of IPEA (global) address is called local address. The substring of IPEA (global) address cut out the local address is called prefix of that local address.

Definition 4. IPEA (global) address with class A top level address is called class A IPEA (global) address, similarly there are class B, C... IPEA (global) address. If a local address is a class A address, it is called class A local address, similarly, there are class B, C... local address.

Definition 5. All the IPEA (global) addresses with same top level address constitute an IPEA domain.

Definition 6. Those nodes with class α local address and with the same address prefix constitute an IPEA cluster, which is a class α cluster, $\alpha \in \{A, B\ldots, Z\}$. The local address is unique in an IPEA cluster. When all variable bits in the local address, i.e. the bits not used to denote the class of the local address, are 0s, this IPEA (global) address denotes the local cluster itself; when all variable bits in the local address are 1s, it denotes all addresses in the local cluster.

Theorem 1. An IPEA (global) address is globally unique.

Proof. As the local address in any IPEA cluster is unique, it is still unique in this IPEA cluster after concatenated with its address prefix. Two local addresses in different IPEA clusters might be same, as their address prefixes are different, so the two IPEA (global) addresses are different.

Corollary 1. IPEA has a bigger address space than IPv4 does.

Proof. Although some bits of every 32 bit fragment of IPEA address are reserved to identify the class of the addresses, which cannot be assigned to identify local address, as an IPEA address is made up of several fragments of address, there can still be more bits to identify global address than IPv4's 32 bits, and the address space can reach IPv6's address space.

Definition 7. The routers (hosts) with class α local address are called class α routers (hosts). If a router's IPEA address is prefix of a class α router's address, it is called that class α router's uplink router. The IPEA cluster where the uplink router resides is called adjacent cluster of that class α IPEA cluster where the class α router resides.

Definition 8. The maximum number of the fragments in an IPEA (global) address in a network is called the number of layers of this network.

Definition 9. The cluster with IPEA (global) address with only one fragment of address is called 0th-level IPEA cluster, similarly, there are 1st, 2nd... level IPEA cluster.

Definition 10. The router in a 0th-level IPEA cluster and linked with IPv4 routers is called IPEA domain border router.

Definition 11. The IPEA router linked with routers in adjacent clusters is called IPEA cluster border router, the other routers in the same cluster are called cluster internal routers.

Definition 12. The stretch factor of address, s, is the ratio of the length of the address to the address actually needed.

So $s_{IPv4} = 32/\lceil \log_2 |N| \rceil$, where |N| is the number of nodes in the network. $s_{IPv6} = 128/\lceil \log_2 |N| \rceil = 4 * s_{IPv4}$. IPEA with k layer address, $s_{IPEA} = k * 32/\lceil \log_2 |N| \rceil$, it is safe to assume that IPEA does not involve more than 5 layer address, then $s_{IPEA} = 3 * 32/\lceil \log_2 |N| \rceil = 3 * s_{IPv4}$.

Corollary 2. IPEA address is more compact than IPv6 address.

4 IPEA Packet Format

The format of extension address header is similar to the format of IPv6 extension header [5], but is aligned by 32 bits (Figure 1).

Next header	Address type and length	Extension address
Extension address		Padding

Fig. 1. Format of extension address header

Table 1.

Type of extension address	Protocol
0	Reserved
1	IPv4 address
2	IPX address
3	NetBIOS name

The value for next header field is protocol numbers defined in RFC1700, with a new number 101 which identifies IPEA extension address, to identify the type of next header or payload. The highest bit in address type and length field is 0 for the source extension address, 1 for destination extension address. The succeeded 4 bits identify types of extension address, the numbers are shown in Table 1. The last three bits identify the length of extension address header in 32 bit words. All destination extension address extension headers are placed after source extension address extension headers.

IPv4 packet format with extension address is illustrated in figure 2, when extension address presented, the protocol field is set to 101. The value of the version field is still 4, so that IPv4 routers and hosts can still process the new packet format. IPv4 packet and IPEA packet are differentiated only by protocol field, i.e. whether there's an extension address header present.

If IPEA packet is fragmented in the intermediate router, there are two options: 1) with nontransparent fragmentation mode, extension addresses are duplicated in every fragments, and the fragments are reassembled in the destination host, which demands that the intermediate routers are extension address sensitive; 2) with transparent fragmentation mode, extension addresses don't have to be duplicated in every fragments, the fragments are reassembled in the border router of the IPEA AD, then the assembled packets are forwarded to destination IPEA host in nontransparent mode.

Version	IHL	Type of service	Total length
ID			Fragment offset
TTL		Protocol	Header checksum
Source address			
Destination address			
Options			
Extension address header			

Fig. 2. Format of IPEA packet header

5 IPEA Routing Algorithm, Proof of Correctness, and Complexity Analysis

5.1 IPEA Routing Algorithm

5.1.1 IPEA Intra-cluster Routers' Update Algorithm

By RIP or OSPF protocol, the intra-cluster routers periodically propagate routing message to its adjacent intra-cluster routers with local address, receive routing message from them, and maintain an intra-cluster routing table.

5.1.2 IPEA Cluster Border Routers' Update Algorithm

1) Perform intra-cluster routers' update algorithm;
2) The border router periodically propagates keep-alive message and its downlink message to its uplink router with IPEA (global) address, and maintains an inter-cluster routing table. If no keep-alive message is received from the downlink router in three periods, the entry for that downlink router will be deleted from the inter-cluster routing table.

5.1.3 IPEA Domain Border Routers' Update Algorithm

1) Perform intra-cluster routers' update algorithm;
2) With BGP protocol, exchanges routing message with adjacent routers in IPv4 domain with IPv4 address, maintains the IPv4 domain routing table.

5.1.4 IPEA Intra-cluster Routers' Forwarding Algorithm

1) The router checks the source and destination address of the packet, to determine that its own IPEA address is in the source address string or in the destination address string, so as to determine to route the packet by the source address string or by the destination address string in the packet;

2) The router decides which fragment of the address string that it should process according to the router's class;
3) Routes the packet with RIP or OSPF protocol.

5.1.5 IPEA Cluster Border Routers' Forwarding Algorithm

1) The router checks the source and destination address of the packet, to determine that its own IPEA address is in the source address string or in the destination address string, so as to determine to route the packet by the packet's source address string or by the packet's destination address string;
2) The router decides which fragment of the address string that it should process according to the router's class;
3) If the destination address of the packet is not in the local cluster, the packet is forwarded to adjacent cluster; otherwise the packet is routed with RIP or OSPF protocol.

5.1.6 IPEA Domain Border Routers' Forwarding Algorithm

1) The router checks the source and destination address of the packet, to determine that its own IPEA address is in the source address string or in the destination address string, so as to determine to route the packet by the packet's source address string or by the packet's destination address string;
2) If the destination address of the packet is not in this IPEA domain, a) if the source and destination addresses of the packet are both IPv4 addresses without extension address, the packet is routed according to IPv4 routing algorithm (RIP, OSPF or BGP), end of the algorithm; b) the packet is forwarded to a NAT proxy [7], where the packet is encapsulated in an IPv4 packet [17] with which the addresses are set to the top-level addresses of the IPEA addresses of the original packet, and is done protocol and address translation, the new packet is routed according to IPv4 routing algorithms (RIP, OSPF or BGP), end of the algorithm.
3) If the destination address of the packet is not in the local cluster, the packet is forwarded to adjacent cluster; otherwise this packet is routed with RIP or OSPF protocol.

5.2 Correctness Proof of IPEA Routing Algorithm

Theorem 2. If there exists a path between two nodes, IPEA routing algorithm can search it out.

Proof. In case of the two nodes in the same IPEA cluster, as RIP or OSPF protocol is used for intra-cluster routing, the correctness of RIP or OSPF routing protocol guarantees the proposition true. In case of that the two nodes are in different IPEA clusters, as every IPEA cluster border router periodically propagates link state to its uplink routers, which guarantees the inter-cluster connectivity is aware, and the link between two border routers is point-to-point, no loop can be there, so the case with two nodes in different clusters is identical to the case with two nodes in the same cluster. In case of that the two nodes are in different IPEA AD (autonomous domain), as the inter-AD routes are generated with BGP, whose correctness guarantees the proposition true.

Theorem 3. IPEA routing path is loop-free.

Proof. An IPEA routing path consists of intra-cluster paths, inter-cluster paths, and inter-domain paths. As every inter-cluster route has only one hop, and the cluster

changes by every hop, so inter-cluster path is loop-free. While intra-cluster path and inter-domain path are generated by RIP, OSPF, or BGP protocol, the loop-avoidance mechanism of these routing protocols keeps the intra-cluster path and inter-domain path loop-free.

Theorem 4. IPEA routing algorithm converges in finite time in absence of network topology changes.

Proof. The IPEA routing algorithm consists of point-to-point inter-cluster routing, RIP, OSPF, and BGP routing algorithm, the convergence mechanisms of RIP, OSPF and BGP routing algorithm guarantee this property.

5.3 Complexity Analysis of IPEA Routing Algorithm

If average number of neighbors of every node in an IPEA cluster is k, as an IPEA router propagates routing message only to its adjacent nodes in the same cluster (the border router to its adjacent border routers), the time complexity of computation taken in a node is O(k). If it spends one unit of time to process one routing message, the time complexity in the worst case is O(|N|), |N| is the number of nodes in the local cluster.

6 Interoperability

As illustrated in previous sections, IPEA is a superset of IPv4, an IPv4 packet is equivalent to an IPEA packet without extension address, a system (a host, a router or a network) that supports IPEA will support IPv4 as well. The problem is how IPv4 network should deal with IPEA packet. With the traffic between two nodes in different IPEA domains, when either the source address or the destination address of the packet is IPEA address with extension address, the packet is encapsulated in an IPv4 packet and done protocol and address translation by an IPEA domain border NAT box in the middle, to traverse IPv4 clouds (as indicated in section 5.1.6).

The interoperability between IPEA and IPv6 is similar to that between IPv4 and IPv6.

7 Performance Analysis

As the routing table in each node has an entry only for each node in the local cluster (in the border node there are in addition several entries for border nodes in adjacent clusters), the routing table length is the size of local cluster. When the network is partitioned into c clusters, the average routing table length stretch factor is 1/c. Routing table length is proportional to the size of cluster and independent of the clustering structure of the network.

Routing path length is proportional to the number of layers of the clustering structure of the network. Load balance of a cluster is dependent on the number of sub-clusters that this cluster has. An optimal clustering structure which is similar to the one in [11] is arrived with a trade-off between the routing path length stretch and the load balance. The detailed account we leave to another paper.

8 Simulation Results

The simulation is done in ns-2 [16] environment, we rewrite the hierarchical routing module, DV, to implement IPEA routing protocol with RIP routing algorithm.

Definition 13. Convergence time of an update is such a time interval, from a network event (a link disconnects or a routing table entry ages) happened until the last routing table has been updated in the network.

Definition 14. Communication overhead of an update is the all routing message traffic in the network in the Convergence time of an update.

Definition 15. The stretch factor of IPEA routing path length, s_p, is the ratio of average IPEA routing path length to average routing path length generated by non-clustered DV algorithm, in the networks generated by the same Waxman random graph model [24].

The objective of our clustering of the network is mainly concerning extending address space and management of networks, instead of optimizing the clustering structure of the network to gain the most reduction in computing, storage, and communication overhead in each node in the network [11]. So in our experiments, the clustering is randomly generated, the link degree between nodes is generated by Waxman random graph model [24]. The bandwidth of the link is set to 100kbps, the period for nodes to propagate routing message is 2 seconds. With different numbers of nodes and different clustering structure of the network, the convergence time, communication overhead, and routing path stretch factor are observed and compared among IPEA and RIP with IPv4 and IPv6 address respectively (Figure 3, 4, 5).

The results of the simulation show, due to partition of the network, the number of nodes in every cluster is reduced, so the computing, storage, and communication overhead taken for maintaining routing tables in each node is reduced. As the number of nodes in the network increases, the reduction of the maintaining overhead of routing tables is more significant, while the increase of the routing path length becomes more trivial, which is consistent with Kleinrock et al's results [11].

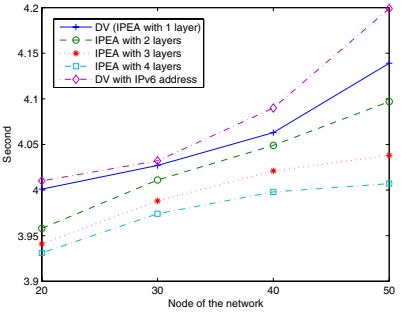


Fig. 3. Number of nodes vs. average convergence time

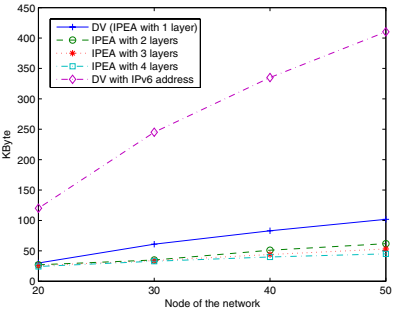


Fig. 4. Number of nodes vs. average communication overhead

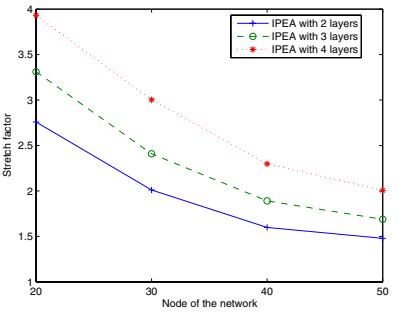


Fig. 5. Number of nodes vs. stretch factor

Compared with IPv6, IPEA is more interoperable with IPv4. IPEA address scales better, with the communication between nodes in the same cluster, only 32 bit address is deployed, while always 128 bit address in IPv6. The size of IPEA routing table does not exceed the size of IPv4 routing table, which helps dealing with routing table explosion. The functional enhancements of IPv6 come mainly from its introduction of extension headers, though only address extension header is introduced into IPEA here, the other extension headers of IPv6 can be introduced into IPEA as well without resulting in more incompatibility between IPEA and IPv4.

9 Conclusion

If IPEA is supported in an AD, the TCP/IP protocol stack in the routers and hosts in this AD has to be updated; the routine of extracting addresses from the packet has to be modified, while with routing algorithm there is nothing special. The transition from IPv4 to IPEA is mostly in software update, the cost should be comparatively low.

IPEA has such advantages: 1) it provides a solution to the problem of address space exhaustion in the Internet; 2) it reduces the routing table length, helps dealing with routing table explosion; 3) little change on the IPv4 addressing scheme, easy for transition; 4) it makes the autonomous network more manageable as using private address.

The modification of protocols concerning IPEA, e.g. ICMP, DNS, etc, is not included in this paper.

References

1. Balakrishnan, H., et al.: A Layered Naming Architecture for the Internet. SIGCOMM, Portland OR, Aug. 2004
2. Bradner, S., A. Mankin: The Recommendation for the IP Next Generation Protocol, RFC1752, January 1995
3. Castineyra, I., N. Chiappa, and M. Steenstrup: The Nimrod routing architecture, RFC 1992, August 1996
4. Cohen, D.: On Names, Addresses and Routings, IEN23, IEN31, ISI, 1978

5. Deering S., R. Hinden: Internet Protocol, Version 6 (IPv6) Specification, RFC2460, December 1998
6. Droms, R.: Dynamic Host Configuration Protocol, RFC2131, March 1997
7. Egevang, K., P. Francis: The IP Network Address Translator (NAT), RFC1631, May 1994
8. Francis, P.: A near-term architecture for deploying PIP. IEEE Network, 7(6), 1993. 30-27
9. Francis, P., R. Gummadi: IPNL: A NAT-Extended Internet Architecture, SIGCOMM'01, August 2001. 69-80
10. Fuller, V., T. Li, J. Yu, K. Varadhan: Classless Inter-Domain Routing (CIDR): an Address Assignment and Aggregation Strategy, RFC1519, September 1993
11. Kleinrock L. and F. Kamoun: Hierarchical Routing for large networks: Performance evaluation and optimization, Computer Networks, Vo 1, 1977. 155-174
12. Lauer, G.S.: Hierarchical routing design for SURAN, Proc IEEE ICC 86, June 1986. 93-102
13. Li, Layuan: The routing protocol for dynamic and large computer networks, Chinese J. computers, 1998(2). 137-144
14. McQuillan, J.: Adaptive routing algorithms for distributed computer networks, BBN Rep. 2831, Bolt Beranek and Newman Inc., Cambridge, MA (1974)
15. Murthy, S. and J.J. Garcia-Luna-Aceves: Loop-free internet routing using hierarchical routing trees, in Proc. IEEE INFOCOM 97, Kobe, Japan, April, 1997
16. ns-2, http://www.isi.edu/nsnam/ns/
17. Perkins, C.: IP Encapsulation within IP, RFC2003, October 1996
18. Ramamoorthy, C.V., W. Tsai: An adaptive hierarchical routing algorithm, Proc IEEE COMPSAC 83, Nov. 1983. 93-104
19. Rekhter, Y., B. Moskowitz, D. Karrenberg, G. de Groot: Address Allocation for Private Internets, RFC1597, March 1994
20. Shoch, J.: Inter-network naming, addressing, and routing, Proc COMPCON 78 Fall, IEEE, 1978. 280-287
21. Tsai, W.T., C.V. Ramamoorthy, W.K. Tsai and O. Nishiguchi: An Adaptive Hierarchical Routing Protocol, IEEE Trans. on Communications, Vol. 38, No. 8, 1989. 1059-1075
22. Turanyi, Z., A. Valko: IPv4+4, 10th International Conference on Networking Protocols (ICNP 2002), November 2002
23. Wang, Z., J. Crowcroft: A Two-Tier Address Structure for the Internet: A Solution to the Problem of Address Space Exhaustion, RFC1335, May 1992
24. Waxman, B.M.: Routing of multipoint connections IEEE J. Select. Areas Commun, 6(9), 1988. 1617-1622

The Star-Pyramid Graph: An Attractive Alternative to the Pyramid

N. Imani and H. Sarbazi-Azad

IPM School of Computer Science and Sharif University of Technology,
Tehran, Iran
navid_imani@softhome.net,
azad@{ipm.ir, sharif.edu}

Abstract. This paper introduces a new class of interconnection networks named *Star-Pyramid*, *SP*(*n*). A star-pyramid of dimension *n* is formed by piling up star graphs of dimensions 1 to *n* in a hierarchy, connecting any node in each *i*-dimensional star, $1< i \leq n$, to a node in $(i-1)$-star whose index is reached by removing the *i* symbol from the index of the former node in the *i*-star graph. Having extracted the properties of the new topology, featuring topological properties, a simple routing algorithm and Hamiltonicity then we compare the network properties of the proposed topology and the well-known pyramid topology. We show that the star-pyramid is more fault-tolerant and has less network diameter than its alternative, the pyramid. Finally, we propose a variant of star-pyramid, namely the generic star-pyramid as a topology with better scalability, fault-tolerance, and diameter.

1 Introduction

An interconnection network can be represented as an undirected graph where a processor is represented as a node, and a communication channel between processors as an edge between corresponding nodes. Tree, mesh, hypercube, pyramid, and star graph are popular interconnection networks. Measures of the desirable properties for interconnection networks include degree, connectivity, scalability, diameter, fault tolerance, and symmetry. There is a trade-off between degree, related to hardware cost, and diameter, related to transmission time of messages [2].

A pyramid network is a hierarchy structure based on meshes. A pyramid of *n* levels, denoted as P_n, consists of a set of nodes $V(P_n)=\{(k, x, y) \mid 0 \leq k \leq n,\ 1 \leq y \leq 2^k\}$. A node $(k, x, y) \in V(P_n)$ is said to be a node at level *k*. All the nodes in level *k* form a $2^k \times 2^k$ mesh. A node $(k, x, y) \in V(P_n)$ is connected within the mesh at level *k*, to four nodes $(k, x-1, y)$, if $x>1$, $(k, x, y-1)$, if $y>1$, $(k, x+1, y)$, if $x < 2^k$, $(k, x, y+1)$, if $y < 2^k$. It is also connected to the nodes $(k+1, 2x-1, 2y)$, $(k+1, 2x, 2y-1)$, $(k+1, 2x-1, 2y-1)$ and $(k+1, 2x, 2y)$ for $0 \leq k < n$ in the next level, $k+1$, and to node $(k-1, (x-1)/2+1, (y-1)/2+1)$ in level $k-1$ [3, 4].

The best route between two nodes is reached using the vertical edges between levels, so the diameter for an *n*-pyramid is $2n$ which is rather high for a network of $(4^n-1)/3$ nodes, leading to high overall costs. The routing algorithm for the pyramid graph, in the worst case, takes the vertical path to the base vertex to bypass the long

T. Srikanthan et al. (Eds.): ACSAC 2005, LNCS 3740, pp. 509 – 519, 2005.

path, otherwise needed to be taken in the mesh. Although this routing schema results in a shorter path, routing between nearly any two nodes is dependant upon the safety of the nodes in higher levels, which makes the network vulnerable to faults while imposing a great deal of congestion on the nodes close to the base node.

In this paper, we introduce a new interconnection network to compensate for the shortcomings of the pyramid, namely large diameter, vulnerability to failure and congestion. The suggested network has attractive properties such as a simple routing algorithm, fault-tolerance, and lower network cost than the pyramid and its variants.

2 The Star-Pyramid: Definition and Basic Properties

An *n-dimensional star graph*, also referred to as *n-star* or S_n, is an undirected graph consisting of $n!$ nodes (vertices) and $(n-1)\ n!/2$ links (edges). Each node is uniquely assigned a label $x_1\ x_2 \ldots x_n$, which is the permutation of n distinct symbols $\{x_1, x_2, \ldots, x_n\}$. Two nodes are joined by an edge *along* dimension d if the label of one node can be obtained from the other by swapping the first symbol and the d^{th} symbol, $2 \le d \le n$. Without loss of generality, throughout we let these n symbols be $\{1, 2, \ldots, n\}$ [1].

The n-dimensional star also called n-star is a node-symmetric and edge-symmetric graph consisting of $n!$ nodes and $n!(n-1)/2$ edges. Each vertex in an n-star has n - 1 incident edges. The network diameter of the n-star equals $\lfloor 3(n-1)/2 \rfloor$.

Definition 1: An n-star-pyramid, SP_n, is constructed by piling up the star graphs of dimensions 1 to n in a hierarchy $(S_1 S_2 \ldots S_n)$ with each S_i ,$1 < i < n$, connected to S_{i-1} from the top and to S_{i+1} from the bottom using some extra links defined by a mapping function $\Psi_{k+1} : V(S_{k+1}) \longrightarrow V(S_k)$ from vertex set of S_{k+1} to the vertex set of S_k. In the context of this paper, we also refer to S_i as the i^{th} level or i^{th} layer alternatively. Hence, the vertex set and the edge set of SP_n can be formally defined as

$$V(SP_n) = \sum_{i=1}^{n} V(S_i)\,,\ E(SP_n) = \sum_{i=1}^{n} E(S_i) \bigcup \sum_{i=1}^{n-1} L_i$$

$$L_i = \{(v_i, v_j) \mid v_i \in V(S_i), j = i+1, v_j \in V(S_{i+1}), \Psi(v_j) = v_i\}$$

where L_i denotes the vertical links connecting the i^{th} level to the $(i+1)^{th}$ level.

Let $T_j = <t_1 t_2 \ldots t_j>$ denote the index of any given node $v_j \in V(S_j)$, in this context we define Ψ_j as:

$$\Psi_j = \{(v_j, v_i) \mid T_j = < t_1 t_2 \ldots t_{k-1} t_k t_{k+1} \ldots t_j >, t_k = j, T_i = < t_1 t_2 \ldots t_{k-1} t_{k+1} \ldots t_j >, j = i+1\}$$

The definition implies that the n-star-pyramid consists of star graphs of dimensions 1 to n with a node v_i in star graph of level i connected to a node v_j of star graph of level $i+1$ if and only if the index of v_j is derived by stuffing of $i+1$ symbol in any of $n+1$ possible positions in the v_i index.

The n-star-pyramid, SP_n, consists of star graphs of all dimensions from 1 up to n. So the number of nodes can directly derived as the sum of the all nodes in all star graphs, i.e. a SP_n has $(1! + 2! + \ldots + n!)$ nodes.

To determine the degree of a node v residing in an intermediate level, say i, one should notice that v is connected to other $i-1$ nodes within the structure of the i-star, v is also linked to the $(i-1)$-Star via a single vertical link. According to the definition of Ψ_j there can be j different nodes corresponding to j different positions (1,2,...,j) for the t_k ($=j$) in T_j, all of which are mapped to the same T_{j-1} so v is connected to $(i+1)$-star via $(i+1)$ links.

The degree of the nodes in the intermediate levels increases by order of two as the level increases, because the last level (level n) doesn't have downward links, the node with the maximum degree is a node in S_{n-1}. The degree of any node v in any intermediate level i is $D(v) = (i-1) + 1 + i + 1 = 2i + 1$. Network degree of $SP_n = 2(n-1) + 1 = 2n - 1$ contributing to a node in the $(n-1)^{th}$ level.

There are two kinds of edges present in SP_n, internal edges in the structure of stars and vertical edges connecting the immediate levels, S_i has $(i-1)i!$ edges so the second term in the equation contributes to the internal edges. Thus, the edge set size in the star-pyramid network is given by

$$| E(SP_n) | = \sum_{i=2}^{n} i! + \sum_{i=2}^{n} \frac{(i-1)i!}{2} = \sum_{i=2}^{n} \frac{(i+1)!}{2}$$

3 Routing

In this section, we propose a simple routing algorithm, for the defined topology and present an upper bound for the distance between two nodes in SP_n as well as the network diameter; finally we derive a formula for the bisection width of the network.

Suppose that our goal is to find a path between two arbitrary nodes, say v_i and v_j, where v_i is a node in level i of the star-pyramid and v_j is a node in level j and $i > j$.

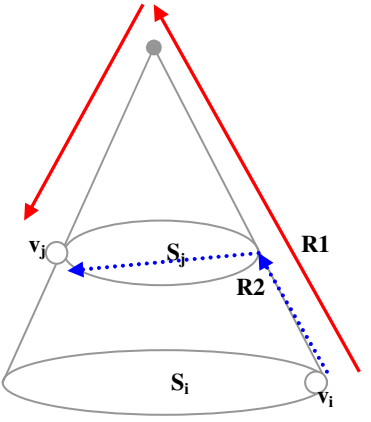


Fig. 1. Routing algorithm in SP_n

Generally, two approaches come to mind for traversing the network from v_i to v_j or vise versa. One can take only the vertical links connected to the lower level to reach the base node then descend the appropriate vertical links until reaching v_j (path $R1$ in fig.1.). In this case, the length of the path will be $i + j - 2$. In the second approach it is necessary that the message move to level j^{th} and from there to take the horizontal path within the star graph in level j^{th} to reach v_j instead of taking the path to the base node (path $R2$ in fig.1.).

In order to obtain the routing algorithm, the length of the two paths $R1$ and $R2$ must be compared. Regardless of which approach is chosen, taking the vertical path from v_i to a node in j^{th} level is inevitable. Therefore, the second part of the route is the determinant factor for the length of the path. Once we are in the same dimension, the problem amounts to choosing whether to travel to the root or take a path within the star graph of that level.

Lemma 1: For any two given nodes v_i and v_j in level m of the SP_n let R_s and R_b denote the shortest path within the star graph and the path taken to the base node respectively, then R_s is shorter than R_b.

Proof: Let $D(R\,;\, v_i,\, v_j)$ be the distance between v_i and v_j taking the path R .
The diameter of a star graph at level m, S_m is $3(m-1)/2$ which is an upper bound for the length of a path between two nodes in the star graph in level m of SP_n, thus

$$D(R_s\,;\, v_i, v_j) \le 3(m-1)/2. \qquad (1)$$

Traveling to the root results in a path with the length of $2(m-1)$ or more formally,

$$D(R_b\,;\, v_i, v_j) = 2(m-1). \qquad (2)$$

From (1) and (2), we have $\forall m>1,\ D(R_s;\, v_i,\, v_j) < D(R_b;\, v_i,\, v_j)$.

Lemma 2: Let R_v denote the vertical path needed to be taken as a part of the path from u to w where $u, w \in V(SP_n)$, $u \in V(S_i)$, $w \in V(S_j)$ then $R_v = \langle v_i v_{i+1} \ldots v_j \rangle$, $\Psi_k(v_k) = v_{k-1}$, $j \ge k > i$, $v_i = u$.

Proof: To obtain the vertical route from u to w we build the R_v iteratively using a constructive method. The vertical links are defined by the function Ψ. Let I_p denote the index of a given node v_p. So at any step, say while in node v_k, $\Psi_k(v_k)$ is appended to R_v as the next node in the path which is the node whose index derived by deletion of the symbol k from I_k. After $i - j$ steps v_j have been chosen, while I_j is reached by deletion of symbols $\{i\,,\, i-1\,,\ldots,\, j+1\}$ from I_i, it can be concluded that $|\, I_j \,| = i - (i-j) = j$ and v_j is a node in level j and R_v is the desired vertical path.

Theorem 1: The path from v_i to v_j is derived by traversing the R_v route built as discussed in the lemma 4.1.2 and then taking the R_s path which is directly obtained from the minimal routing on the S_i, the detailed routing algorithm is as follows:

```
Routing Algorithm (SP_n)
Input:  I_s index of s ∈ V(S_i) as source, and
        I_d index of d ∈ V(S_j) as Destination.
Output: R_m the path between s , d.

{  R_m = s;
   if (I_s = I_d) then return R_m;
   else
      if (|I_s|=|I_d|) then return  R_m.StarPath (I_s , I_d);
      else return  R_m.SP_n-Routing(Delete(I_s,i),I_d);
}
```

Thus, according to this routing algorithm, the vertical link edges from node v_i must be taken to reach a node in level v_j, a node whose index is equal to the string obtained by deletion of all the symbols $j + 1 \ldots i$ from the index of node v_i without changing the order of other symbols. Once in level j, the minimal path in S_j must be taken to reach v_j.

The star graph is a topology with bidirectional edges, so generally routing from v_i to v_j or vise versa leads to the same set of nodes for the path. Releasing the $i \geq j$ constraint, the upper bound for the length of the path would be derived as

$$3(\min(i, j)-1)+|i-j|.$$

Theorem 2: Diameter (SP_n) = Diameter (S_n) = $3(n-1)/2$.

Proof: Given a star-pyramid graph SP_n the diameter of a graph is defined as the maximum distance of any two vertices of the graph calculated on the minimum path between the nodes. Using the routing algorithm proposed in the Theorem 4.1.3, the two nodes with maximum distance are located in the n^{th} level star graph. Thus the diameter of a star-pyramid graph is $3(n-1)/2$.

The bisection width of a network is the minimum number of links that have to be removed to partition the network into two (disjoint) halves. It is an important parameter for interconnection networks and is crucial to their cost and performance. However, this parameter is quite difficult to obtain for many networks. In what follows, we will show that tight bounds on the bisection width for star pyramid graph based on the bisection with of the star graphs using the following.

Postulate 1: The bisection width of an N-node star graph is equal to $1/4N + O(N)$ [7].

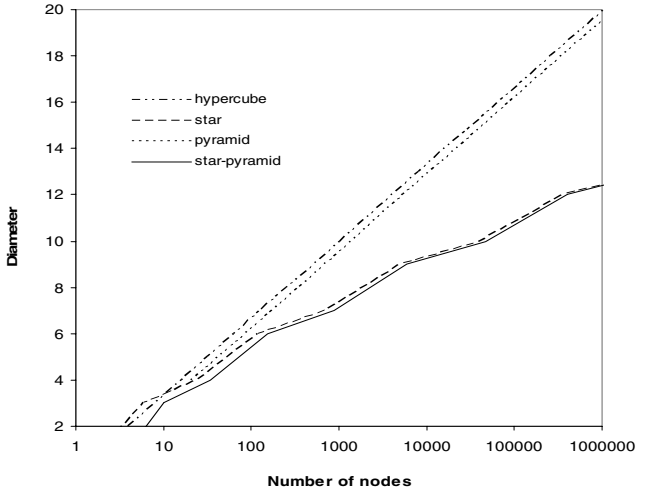
Theorem 3: The bisection width of a star-pyramid equals

$$\sum_{i=2}^{n} BW(S_i) = 1/4\sum_{i=1}^{n} i! + o(N^2)$$

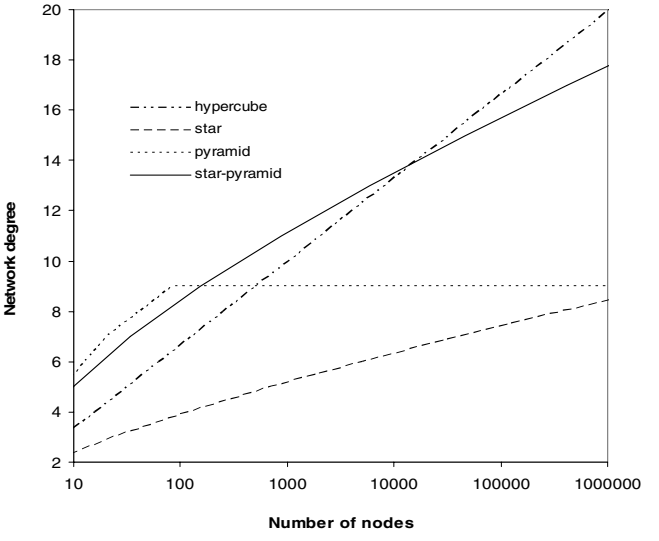
Proof: By definition, the Bisection Width of a network is the number of edges needed to be cut to divide the network in to two sub-networks of equal (or nearly equal if |V| is odd) nodes. It is easily proved that

$$\sum_{i=1}^{n} i! < \sum_{i=1}^{n} n! < (n+1)!$$

Therefore, even if we cut the links between n^{th} and $(n-1)^{th}$ level, the two pieces wouldn't have equal nodes. So the only way to cut the network is to cut all levels (except for the 1^{st} level) into two pieces using a vertical plane. As a result of the Tree structure of the vertical edges in the SP_n ($StarTree_n$), these edges do not form a loop and hence no vertical edges will need to be cut and the theorem holds as such.



(a)



(b)

Fig. 2. The network diameter and degree of SP and other important network topologies as a function of network size

4 Comparison to Other Networks

As it was mentioned before, star-pyramid due to the robust architecture of the star graph in its layers is deemed to perform better than pyramid. For example as a result of the sub-logarithmic diameter of the star graph, the diameter of the network does not exacerbate in the presence of vertical nodes, and the overall diameter of the network is proved to be equal to the diameter of its lower star layer.

A comprehensive study of the star-pyramid and of the two networks when drawn against the number of nodes (in a logarithmic scale), as shown in fig.2. also reveals the significant superiority of the star-pyramid over pyramid and other important network topologies including star graph, hypercube, in terms of the network diameter. Much of the popularity of the star graph is because of its low sub-logarithmic diameter; it is while star pyramid, as it is apparent from the figure, shows even smaller diameter than the star graph. As can be seen in the figure, the network degree of the star-pyramid is better than the equivalent hypercube, but worse than equivalent star graph and pyramid network.

5 The Generic Star-Pyramid

As it has been shown in the routing algorithm, the chosen path between two nodes is within the star graph when the two nodes are in the same level. This implies that removing some of the top level can be done safely without being worried about the routing algorithm.

Definition 2: A *Generic Star-Pyramid* (m,n), $GSP_{m,n}$, is a graph obtained by removing the upper SP_{m-1} component from SP_n ,where $m \leq n$. It thus consists of only levels m to n of SP_n. The definition implies that a star-pyramid is a special form of the generic star-pyramid having $m = 1$. While $GSP_{k,k}$ consists of only a S_k., $GSP_{1,n} = SP_n$, $GSP_{n,n} = S_n$.

Theorem 4: The following equations hold for $GSP_{m,n}$:

$$|V(GSP_{m,n})| = \sum_{i=m}^{n} i!$$

$$|E(GSP_{m,n})| = \sum_{i=m+1}^{n} i! + \sum_{i=m}^{n} \frac{(i-1)i!}{2}$$

$$Degree(GSP_{m,n}) = 2n-3$$

$$Diamater(GSP_{m,n}) = 3(n-1)/2$$

$$BSW(GSP_{m,n}) = \sum_{i=m}^{n} BSW(S_n) \cdot$$

The routing algorithm for the generic star-pyramid is the same as that proposed for the star-pyramid.

6 Topological Properties

One of the attractive properties of the pyramid class of graphs is that the same graph topology of a lower dimension is observable within the graph, this property yields to a simple recursive definition for the graph while preserving a great degree of simplification and conciseness in the formulas derived for the different properties of the network. For example each pyramid P_n consists of a based node connected to 4 P_{n-1} components. The recursive structure for a star-pyramid is not as straightforward as it is in the pyramid.

Definition 3: Let S_n denote an n-star-graph, then S_n consists of n S_{n-1} substars, S_{n-1}^{k}. In particular, the index of S_{n-1}^{k} is in the form of $<p_1\, p_2 \ldots k>$, $1 \le k \le n$ [5].

Theorem 5: $GSP_{m,n}$ consists of m $GSP_{m-1,n-1}$ and an $GSP_{m,n-1}$ components.

Proof: $GSP_{m,n}$ by definition consists of an i-star-graph in its i^{th} level. An i-star graph can be decomposed to i S_{i-1}^{k} substars having node indices of the form $p_1\, p_2 \ldots p_{i-1}\, k$, $k \in \{1 \ldots i\}$ where k is a fixed number for each substar. Based on the definition of Ψ, adjacent nodes in the $i+1^{th}$ level can be reached by inserting the $(i+1)$ symbol in the positions 1 to $(i+1)$ of the index of a node in level i.

The same situation also holds for level $(i+1)$. Thus, it consists of $(i+1)$ S_i^k substars, $k \in \{1,\ldots,(i+1)\}$, where indices of the nodes are in the form of $p_1\, p_2 \ldots p_i\, k$. By inserting the $(i+1)$ symbol in positions $1..i$ of the indices of the S_{i-1}^{e} nodes, $e \in \{1..i\}$ in level i, S_i^e is obtained, which is located in $(i+1)^{\text{th}}$ level. The resulting i S_i^e's in level $(i+1)$ can be further inserted to be mapped to S_k^e of higher dimensions. Finally, in level n, i S_{n-1}^{e} results, all of S_k^e, $m-1 \le k < n$ are connected together and form m $GSP^{e}_{m-1,n-1}$ sub-GSP .

On the other hand, inserting $(i+1)$ in the $(i+1)^{\text{th}}$ place (so far, we have had only stuffed $i+1$ in positions 1 to i of the index of the node in level i) in the nodes of S_i in level i results in another S_i^{i+1} substar in level $(i+1)$, which is the missing $(i+1)^{th}$ S_i substar. This substar can be treated as any other normal substar from this stage and can thus be further inserted with symbols, building a $GSP_{m,n-1}$ component. The recursive structure of $GSP_{m,n}$ is depicted in fig.3.

Embedding of cycles in the network topology is one of the interesting problems concerning interconnection networks. A graph is said to be Hamiltonian if there exist a ring embedding scheme of length $|E|$ for the graph. A great deal of work has been done concerning the analysis of the Hamiltonian properties in the pyramid network. Here we show that a generic-star-pyramid is Hamiltonian knowing that this property holds for an n-star graph [6].

Theorem 6: A Hamiltonian cycle can be embedded to a generic-star-pyramid $GSP(m,n)$ traversing at least an edge in level m, $m > 1$.

Proof: Because of the recursive structure of the network, we use the strong induction to prove the theorem. As a result of 8.1, the theorem holds for $GSP(2,3)$ and $GSP(k,k)=S_k$ and we use it as the base of the induction.

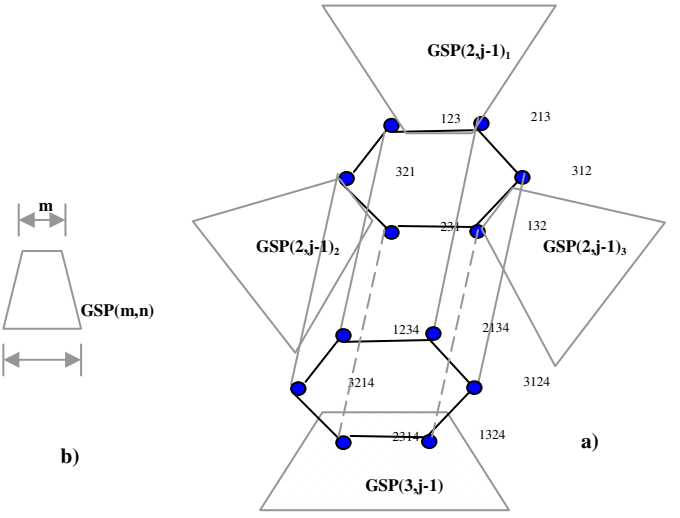


Fig. 3. a) Recursive structure of Generic Star Pyramid (3,j), b) Notation for a GSP(m,n)

Suppose that for each $GSP(i\ ,\ j)$, $i \le m,\ \ j < n,$ there exist a Hamiltonian cycle passing from at least a single edge of S_i in level i. According to the recursive structure of the graph, $GSP(m,n)$ consists of m $GSP(m-1,n-1)$ and a $GSP(m,n-1)$. Assuming that the theorem holds for $GSP(m,n-1)$, in the structure of the S_m in the level m there is an edge, say $e = <u,v>$, which is included in the Hamiltonian cycle built on $GSP(m,n-1)$, disconnecting e results in a Hamiltonian path from u to v within $GSP(m,n-1)$.

For any of m $GSP(m-1,n-1)$, the edge contained in the S_{m-1} can be replaced by a Hamiltonian path (after deleting the immediate link), so building the cycle in S_m of $GSP(m,n)$, and deleting the immediate nodes as described results in a Hamiltonian cycle which passes through all the star-pyramid except for the isometric $GSP(m\ ,n-1)$.

$GSP(m,n-1)$ can also be represented by a loop so we replace the edge in the $m+1$ S_{m+1} with the Hamiltonian path, the only problem which remains is how to merge the cycle of level $m+1$ to level m. As the transform is isometric for $GSP(m\ ,\ n-1)$ there is a one to one correspondence between the nodes in the level m and the $(m+1)^{th}$ $substar_m$ nodes. One idea is to cut the correspondent link to the one which was cut as a part of a Hamiltonian cycle and then connect the 4 nodes using the vertical connections. This builds the Hamiltonian cycle for $GSP(m\ ,\ n)$ while it is simply observable that the Hamiltonian cycle at least passes a single edge. Thus the theorem is proved. The embedding problem in *GSP* is shown in fig.4.

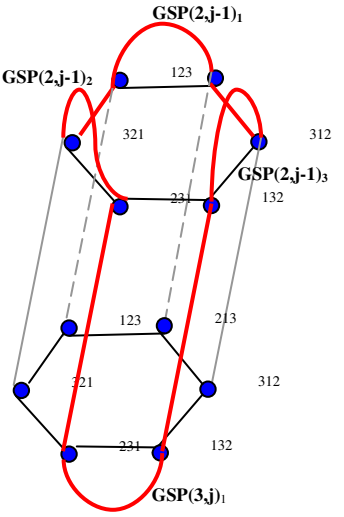


Fig. 4. Embedding a Hamiltonian cycle in the GSP(3,j)

7 Conclusions

The Star-pyramid graph, using the robust topology of the star graph in its levels, provides higher fault tolerant and extremely lower network diameter compared to the pyramid of equal size. To improve fault-tolerance further, a generic form of star-pyramid is proposed. One of the astonishing features of the star-pyramid is that if the network is cut from a level, say i, a generic star-pyramid $(i+1,n)$ and a star-pyramid(i) results, with any of these two graphs transmitting the routing traffic to the nodes in the newly resulting network, exactly the same way as it used to be in the star-pyramid(n). The generic star-pyramid is also more scalable, letting an $m-1$ star or an $n+1$ star to be augmented to the network while obviating the necessity to modify the connections in the rest of the network. The Hamiltonian properties of the proposed networks are quite useful as they can merit designing fault-tolerant communication algorithms for the network.

References

[1] S.B. Akers, D. Harel, B.Krishnamurthy, "The star graph: an attractive alternative to the n-cube", Proc. International Conference on Parallel Processing, St. Charles, Illinois, 1987, pp.393-400

[2] S.B. Akers, B. Krishnamurthy, "A group-theoretic model for symmetric interconnection network", IEEE Trans. Comput., Vol.38, No.4, 1989, pp. 555–565.

[3] H. Sarbazi-Azad, M. Ould-Khaoua, L.M. Mackenzie, "Algorithmic construction of Hamiltonians in pyramids", Information Processing letters, Vol. 80, 2000, pp.75-79.

[4] F. Cao, D.F. Hsu, "Faut-tolerance properties of pyramid networks", IEEE Trans. Comput., Vol.48, 1999, pp.511-527.
[5] D.K. Sakia, R.K. Sen, "Two ranking schemes for efficient computation on the star interconnection network", IEEE Trans. Parallel and Dist. Systems, Vol. 7, No. 4, 1996, pp. 321-327.
[6] Y.C. Tseng, S.H. Chang, J.P Sheu, "Fault-tolerant ring embedding in a star graph with both link and node failure", IEEE Trans. Parallel and Dist. Systems, Vol. 8, No. 12, 1997, pp. 1185-1195.
[7] C.H. Yeh, B. Parhami, "On the VLSI area and bisection width of star graphs and hierarchical cubic networks", Proc. of the 15th International Parallel & Distributed Processing Symposium, 2001, pp.72.

Building a Terabit Router with XD Networks

Huaxi Gu, Zengji Liu, Jungang Yang, Zhiliang Qiu, and Guochang Kang

State key lab of ISN, Xidian University, Xi'an, China 710071
{hxgu, zjliu}@xidian.edu.cn, jgyang_xian@163.com
zlqiu@mail.xidian.edu.cn, gckang@163.com

Abstract. Direct interconnection network has been widely used in supercomputer systems. Recently, it is considered to be used to build terabit router by the industry. This paper presents a distributed and scalable switching fabric based on a new direct interconnection network. It is a scalable topology and can be expanded in two ways easily, thus minimizing the initial investment of service providers. Its distributed control can offer low hardware complexity. Virtual cut through switching is used to achieve high throughput and low latency. The quality of service is guaranteed by introducing virtual channels based on the concept of DiffServ. The fault tolerant and load-balanced routing algorithm can offer deadlock and livelock freedom. It helps the network continue to work even with faulty parts existed. Finally, the performance of the proposed switching fabric is evaluated under various conditions. The results show it can outperform its counterpart in latency and throughput. It achieves terabit throughput with average latency of 1 us or so.

1 Introduction

Rapid growth of the Internet and huge data transfer speed has created a big demand for high performance and large capacity routers. The service providers have to upgrade their backbone routers from gigabit to terabit or even petabit capacity. Hence, one of the most important requirements for the routers is good scalability. The scalability of a router is dictated by the employed switching fabric, which transfers packets from input ports to output ports. To scale a router to handle a larger number of high-speed ports requires a switching fabric that is itself economically scalable. Traditionally, routers have utilized crossbars (e.g. Cisco 12000 Series [1]), or shared buses (Juniper M160 Router[2]) for their switching fabrics. Buses are not scalable to high bit rates, and crossbars, because their cost grows as the square of the number of cross points, cannot be economically scalable to large number of nodes. Recently, direct interconnection networks have become popular architectures for switching fabrics in the core routers. For example, Avici Systems uses 3-D torus as switching fabric in its terabit router AVICI TSR^{TM} [3], while Pluris makes use of hypercube in TeraPlex20 [4]. But these two topologies are not planar, and the trends have moved on to lower-dimensional topologies, which can be expanded more easily. We proposed a scalable XD (cross and Direct) network [5]. It is a planar topology with many advantages such as symmetry, path diversity, short diameter and easy scalability. Based on this topology, we design a new distributed switching fabric for the core routers.

T. Srikanthan et al. (Eds.): ACSAC 2005, LNCS 3740, pp. 520 – 528, 2005.

The rest of the paper is organized as follows: In section 2, we introduce XD network and compare it with other popular topologies. Section 3 presents the design of a terabit router switching fabrics based on XD network, including node architecture, choice of switching mechanism, routing algorithm and quality of service implementation. Section 4 describes the different simulations performed, as well as the results obtained. Finally, we conclude this paper in section 5.

2 XD Network

XD network is a symmetric topology as is shown in Fig.1. For notational simplicity, let $[s]_t$=s mod t for all s∈ I and t∈ I^+, where I is the set of integers and I^+ is the set of positive integers.

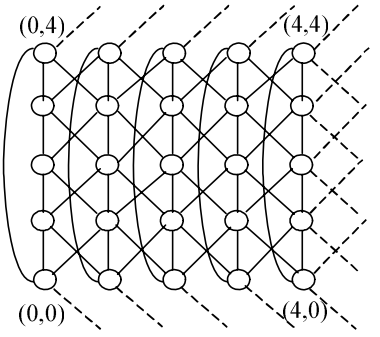


Fig. 1. Illustration of XD network topology

Definition 1. An m × n XD **network** is a directed network $G_{XD}=(\mathbb{N}_{XD}, \mathbb{C}_{XD})$, defined as:

$$\mathbb{N}_{XD}=\{(a,b)\mid\ 0\le a\le m-1,\ 0\le b\le n-1\}$$

$$\mathbb{C}_{XD}=\{\langle(a_u,b_u),(a_v,b_v)\rangle \mid (a_u,b_u),(a_v,b_v)\in \mathbb{N}_{XD}\cap((a_u=a_v\cap b_u=[b_v\pm 1]_n)\cup(a_u=[a_v\pm 1]_m\cap b_u=[b_v\pm 1]_n))\}$$

where $m\ge 2, n\ge 2$, (a,b) represents the coordinate of the node in XD network and (a_u,b_u) (a_v,b_v) are the coordinates of the nodes ***u*** and ***v*** respectively.

A topology is evaluated in terms of a number of parameters. In this paper, we are interested in symmetry, diameter, degree, average distance, scalability and bisection width, which are important aspects of the switching fabrics in the core routers [1-4]. A graph is said to be regular if all the nodes in the graph have the same degree, and homogeneous if all the nodes in that graph are topologically identical. Diameter is the maximum distance in the topology. Average distance is a major component in determining the latency of a network. Interconnection networks should scale economically and incrementally, especially when used as switching fabrics in the terabit routers. The bisection width is defined as the minimum number of links that must be removed to partition the original network into two equal parts.

Table 1 summarizes the degree, diameter, average distance and bisection width of four popular networks, each with N nodes. Details can be found in [5] and [6].

Table 1. Comparisons of the popular networks

Topology / Parameter	2D Torus	3D Torus	Hypercube	XD network
Degree	4	6	$\log_2 N$	6
Diameter	$\sqrt{N}$	$\frac{3}{2}\sqrt[3]{N}$	$\log_2 N$	$\sqrt{N}/2+1$
Average distance	$\sqrt{N}/2$	$\frac{3}{4}\sqrt[3]{N}$	$\frac{1}{2}\log_2 N$	$\frac{1}{3}\sqrt{N}$
Bisection width	$2\sqrt{N}$	$2\sqrt[3]{N^2}$	$\frac{1}{2}N$	$4\sqrt{N}$

XD network is homogeneous and regular. Complete regularity of XD network can lead to a much better performance. This property is useful for practical implementation of switching fabrics because the same routing algorithm can be applied to each node. XD network has a fixed degree 6 and can offer path diversity as most other direct interconnection networks. This feature facilitates designing fault tolerant and load-balanced routing algorithms. XD network has O ($\sqrt{N}$) average distance. It makes use of the crossing and wraparound channels to reach the distant nodes with less hops compared to mesh or torus topology. XD network can offer better scalability than hypercube and 3D tours. For hypercube each node must be configured with ports for the maximum dimensionality of the network, even though these ports are unused in small configurations. For XD network, the smallest extension unit is a row or a column, i.e. a one-dimensional sub graph. In the 3D torus, the smallest extension unit is a plane, i.e. a two-dimensional sub graph. Therefore, the scaling complexity of the XD network is O(n) compared to $O(n^2)$,which is higher.

In general no single topology can provide every desired feature. Considering symmetry, path diversity, scalability and implementability, which are the basic requirements for the fabrics in the routers, XD network is an attractive candidate network topology.

3 Building a Terabit Router with XD Network

3.1 Node Structure

As is shown in Fig 2, the node contains input buffers, a processing unit including CPU and local memory, a virtual channel (VC) allocator and a 16x16 crossbar. The input buffers are used to store flits (fixed size cells [6]). The processing unit takes charge of implementing the routing algorithms and switching mechanism and flow control. VC allocator assigns virtual channels to the incoming flits.

The traffic can come from six aggregated channels and two line cards. There are separate queues for each port, named virtual channel. This queue structure has two usages. One is to guarantee that packets destined for output O_a never block packets

destined for output O_b. The other use is to provide differentiated services. Three sets of virtual channels are provided for gold, silver and copper traffic.

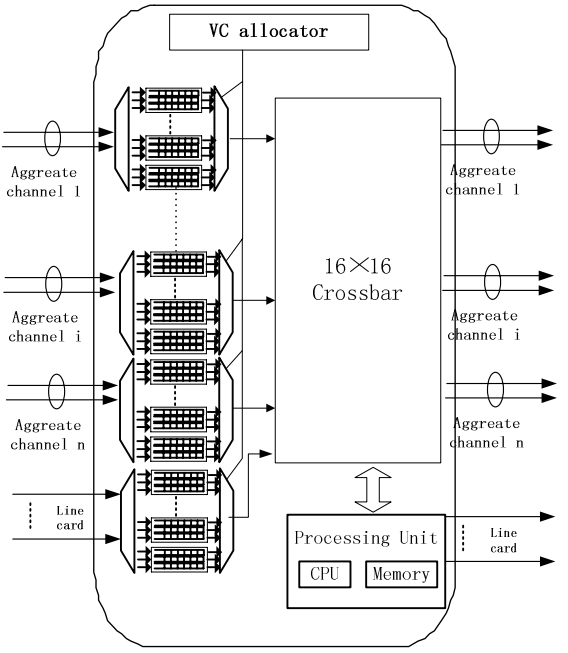


Fig. 2. The node structure of XD network

The nodes in XD networks are connected by aggregated channels, which contain two 10G physical channels. Since each node has six neighbors, 12 of 16 ports of the crossbar are used for internal channels. The remained 4 ports of the crossbar are used for line cards. With such structure, our switching fabric can offer good scalability. It can be expanded to large scale in two ways. One way is to add more line cards to the node in the existing switching fabric. Since there are four ports of crossbar are left for line cards, each node can be loaded with up to 4 line cards. For example, if an 8x8 XD network is used, each node can be configured with one OC192 line card. Thus, the overall switching capacity is 640 Gbit/s. If the traffic grows to the magnitude of terabit, the second line card can be added to each node. Hence, each node has two OC192 line cards and the overall switching capacity scales to 1.28 Tbit/s. In this way, the switching capacity can be doubled without changing the existing topology. Another way is to add rows or columns of nodes to the existing topology. In this way, XD network can offer easier scalability than 3D Torus or hypercube, as discussed in section 2.

3.2 Choice of Switching Mechanisms

Switching mechanism defines how messages pass through the switching fabrics. A variety of switching mechanisms have been proposed [6], among which are: circuit switching (CS), store and forward (S&F), wormhole switching (WS), virtual cut

through switching (VCT) and pipelined circuit switching (PCS). These switching mechanisms are mainly designed for multi-computer systems while the core router presents a somewhat different set of requirements. For example, most commercial multi-computer systems implement WS. But it is not suitable for fabrics in terabit routers because of its relatively low throughput. In our design, we choose VCT as the inner switching mechanism due to the following reasons. Firstly, in most core routers, the switching fabrics internally operate on fixed-size data units. Examples of such routers and switches can be found in both commercial products and laboratory prototypes, such as Cisco GSR [1], the Tiny-Tera [7] and so on. Using fixed-size data units in the switch has many advantages. For example it can make the implementation much easier compared to the variable-length packets. Therefore, VCT, WS and PCS are preferred since they cut packets into fixed size data units. Secondly, the terabit class routers have to handle a large number of high-speed ports. So fabrics may be expanded to large scale. But the delay of S&F is proportional to the distance. Hence, higher delay is obtained for large-scale fabrics if S&F is used. The distance does not heavily affect the latency of WS and VCT, so delay-sensitive real time application will greatly benefit from VCT and WS because of their shorter latency. Therefore, they are suitable for the scalable fabrics in terabit class routers. Finally, considering the heavy traffic faced by the terabit router, VCT is more suitable than WS and PCS. The reason is that VCT can offer lower latency and higher throughput than WS or PCS, which is also been validated in [8].

3.3 Quality of Service

Developing a switch fabric with sufficient Quality of Service (QoS) capability is one of the basic requirements to the designers. We implement a simple mechanism in XD network based on the concept of Differentiated Service (DiffServ). Following the DiffServ philosophy, each node doesn't hold status information about the passing-through traffic. And no per-flow signaling exists in the networks. The nodes handle the traffic in different ways based on the service class. The Olympic traffic class [9] is used, dividing the traffic into three classes, namely gold, silver and copper. Separate virtual channels associated with each port are used for different services. Each class of service only routes in its own virtual networks. A simple WRR strategy is used to share the channel bandwidth among all the virtual channels. Therefore, Gold class service can obtain priority over the other two in competing for channel bandwidth. These policies ensure that Gold class traffic can experience low latency and delay jitter when passing through the fabrics.

3.4 Routing in XD Network

Routing algorithm plays an important role in the performance of the switching fabrics. The algorithm should be deadlock free, livelock free, fault-tolerant and load-balanced. In literature, algorithm designers often focus their attention on one of the features above. For example, some design fault tolerant routing algorithm that cannot provide balanced load. And load-balanced routing algorithms are not fault tolerant. We proposed a distributed load-balanced algorithm for XD network in [10], which is fault tolerant, deadlock and livelock free. The potential deadlocks are resolved by detection and recovery mechanism. Deadlocks can be detected by a simple time-out criterion.

A TwaitCounter(n_c,i) is associated with physical channel i at node n_c. It is incremented every clock cycle. Hence it keeps tracks of the number of cycles during which the node n_c cannot send out the packet in i direction. When TwaitCounter(n_c, i) is greater than the threshold T_{out}, the packet is ejected from the network as a potential deadlocked packet. When the packet is transmitted, either forwarded to the next node or ejected out of the network, TwaitCounter(n_c, i) is reset. Then by using a software-based recovery mechanism, the potential deadlocked packet is absorbed by the local node and will be retransmitted at a later time.

In the case of the occurrence of faulty components, the algorithm can tolerate link or node faults of arbitrary numbers. A fault ring consists of non-faulty channels and nodes that can be formed around each faulty region. The algorithm routes the traffic on the fault rings and bypasses the faulty region. Moreover, only the nodes on the fault rings need to maintain a small amount of fault information. The algorithm can tolerate convex fault regions regardless of possible overlapping of the boundaries of different fault regions. It can balance the traffic and improve the performance of the networks.

4 Simulation Results

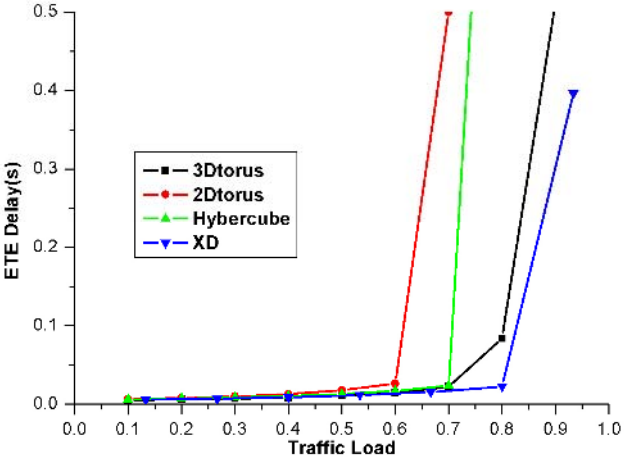
To evaluate the performance of our design, we use one of a powerful software simulation package-OPNET [11]. OPNET provides a comprehensive development environment for the specification, simulation and performance analysis of communication networks.

Each node operates asynchronously. They generate packets at time interval chosen from a negative exponential distribution. Unlike the traditional use of fixed-length packets in simulations [6], we use a specific packet length distributions SP (Size and Percent). It is based on the IP (Internet Protocol) packet size and percentages sampled over a two-week period [12]: 40 bytes (56 % of all traffic), 1500 bytes (23 %), 576 bytes (16.5 %) and 52 bytes (4.5 %). To the best of our knowledge, ours is the first attempt to incorporate such packet length distributions into evaluating performance of direct interconnection networks. Such configurations of simulation environments are more close to reality, which makes the results more convincing.

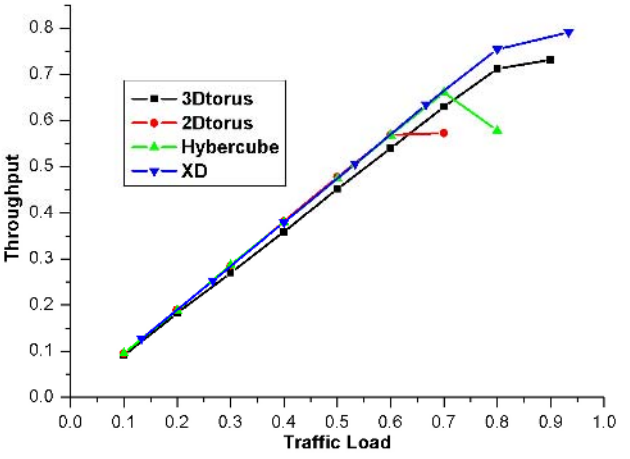
The performance of is measured in terms of ETE (End to End) delay and throughput. The ETE delay is defined as the average time from the packet generation to the time when it reaches the destination. We use normalised throughput, which is equal to the number of packets that can be transmitted at the maximum load.

Firstly, we make comparison of XD network with 2D torus, 3D torus and Hypercube in Fig.3. The three networks are of the same size (N=64). The routing algorithm used is proposed by Duato in [13]. It has been accepted by many real systems such as the Cray T3E [14], Reliable Router [15] and so on. In the case of 3D torus, the algorithm requires at least three virtual channels (VC), which are divided into two classes *a* and *b*. Class *b* contains two virtual channels, in which deterministic routing is applied. The rest virtual channels belong to class *a*, where fully adaptive routing is used. The messages can adaptively choose any virtual channels available from class *a*. If all the virtual channels of class *a* are busy, the message enter channels that belong to class *b*.The results show that XD network

performs better than 2D torus, 3D torus and 6Cube for both heavy and light traffic load. It is the last to saturate and achieves the lowest latency among the three.



(a) ETE delay vs. Traffic load



(b) Throughput vs. Traffic load

Fig. 3. Comparison of XD with 2D torus, 3D torus and 6Cube

Fig 4 shows the performance of XD network as the line rate changes. It is obvious from Fig 4 that the throughput keeps around 99.9% and the latency keeps at the

magnitude of 1 us, when the line rate increases from 4Gbit/s to 20Gbit/s(each node configured with two 10Gbit/s line cards and each works at line rate). When the input traffic is 20Gbit/s into each node, the overall capacity is 20Gbit/s x 64 = 1.28 Tbit/s, that is, XD network has achieved Tbit/s switching. At this point, the average latency is just 1.11us, the throughput keeps at 99.87% and the average link utilization is only 43.462%. Compared to the schemes using multi-stage interconnection networks, the internal speedup of XD network is 1 and still achieves satisfactory performance (Latency and Throughput). What's more, the whole system is working at low load when it achieves terabit switching capacity, which is a satisfactory result for the carrier.

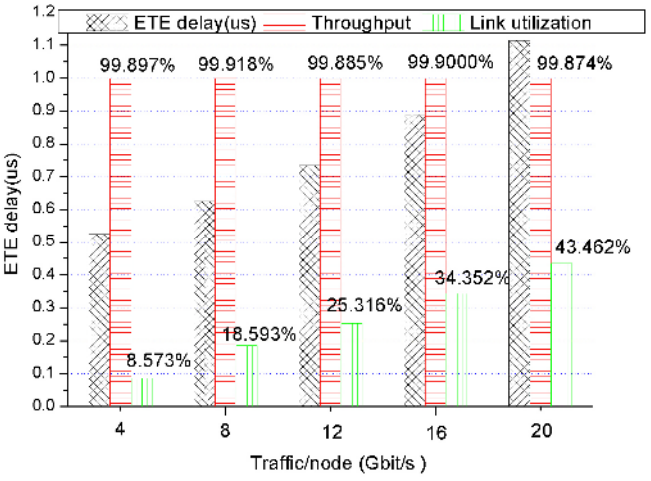


Fig. 4. Performance of the XD network

5 Conclusions and Future Work

This paper introduce a scalable distributed switching fabric architecture based on a new interconnection networks. This architecture can be used to build cost effective switching fabrics supporting terabit class routers. The proposed switching fabric is evaluated by extensive simulation for various sizes. The results show that it outperforms its 2D Torus, 3D Torus or hypercube counterpart for good network performance. It is ideally suitable for scalable routers. Our future work will be focused on the collective communications in this new architecture, such as multicast and broadcast.

Acknowledgment

The authors are extremely grateful to the three anonymous reviewers, who offer us helpful suggestions to improve this paper. This work was supported by the National Science Foundation of China under Grant No.90104012 and the National High-tech Research and Development Plan of China under grant No.2002AA103062.

References

1. Cisco Systems, Cisco 12016 Gigabit Switch Router, Data Sheet. 2001, http://www.cisco.com.
2. Juniper networks, Juniper M160, White paper,2000, http://www.juniper.net/
3. W. J. Dally: Scalable Switching Fabrics for Internet Routers, White paper, Avici Systems Inc. 2001
4. Pluris, Inc., Avoiding Future Internet Backbone Bottlenecks, white papers, On the Web at http://www.pluris.com/. 2000.
5. Z. Qiu H. Gu,et al. A novel scalable topology, China Patent, CN1431808A, 2004
6. W. Dally and B. Towles, Principles and Practices of Interconnection Networks, Morgan-Kaufmann Press, 2004
7. McKeown N., et al., The Tiny Tera: a packet switch core, IEEE Micro Magazine, vol. 17, Feb. 1997, 27-40
8. H. Gu , et al. Choice of Inner Switching Mechanisms in Terabit Router, Lecture Notes in Computer Science, vol. 3420, No.1, pp. 826–833, 2005.
9. J. Heinanen, F. Baker, W. Weiss, and J. Wroclawski. *Assured Forwarding PHB Group*. June 1999. Internet RFC 2597.
10. H. Gu, G. Kang et al, An efficient routing algorithm for XD networks, Technical Report, BNRD-04-018, Xidian University, 2004
11. OPNET Modeler, OPNET Modeler manuals, MIL 3, Inc.3400 International Drive NW, Washington DC 20008 USA,1989-2004.
12. David Newman, Internet Core Router Test, www.lightreading.com. March 6, 2001
13. J. Duato, A New Theory of Deadlock-Free Adaptive Routing in Wormhole Networks, IEEE Trans. on Parallel Distrib. Sys vol. 4, 1993,1320-1331
14. S. L. Scott and G. M. Thorson. The Cray T3E Network: Adaptive Routing in a High Performance 3D Torus. In Proc. Symp. High Performance Interconnects (Hot Interconnects 4), pages 147--156, August 1996.
15. W. J. Dally, et al., The Reliable Router: A reliable and high-performance communication substrate for parallel computers, Proceedings of the Workshop on Parallel Computer Routing and Communication, May 1994, 241--255

A Real Coded Genetic Algorithm for Data Partitioning and Scheduling in Networks with Arbitrary Processor Release Time

S. Suresh[1], V. Mani[1], S.N. Omkar[1], and H.J. Kim[2]

[1] Department of Aerospace Engineering,
Indian Institute of Science, Bangalore, India
{suresh99, mani, omkar}@aero.iisc.ernet.in
[2] Department of Control and Instrumentation Engineering,
Kangwon National University, Chunchon 200-701, Korea
{khj}@kangwon.ac.kr

Abstract. The problem of scheduling divisible loads in distributed computing systems, in presence of processor release time is considered. The objective is to find the optimal sequence of load distribution and the optimal load fractions assigned to each processor in the system such that the processing time of the entire processing load is a minimum. This is a difficult combinatorial optimization problem and hence genetic algorithms approach is presented for its solution.

1 Introduction

One of the primary issues in the area of parallel and distributed computing is how to partition and schedule a divisible processing load among the available processors in the network/system such that the processing time of the entire load is a minimum. In the case of computation-intensive tasks, the divisible processing load consists of large number of data points, that must be processed by the same algorithms/programs that are resident in all the processors in the network. Partitioning and scheduling computation-intensive tasks incorporating the communication delays (in sending the load fractions of the data to processors) is commonly referred to as divisible load scheduling. The objective is to find the optimal sequence of load distribution and the optimal load fractions assigned to each processor in the system such that the processing time of the entire processing load is a minimum. The research on the problem of scheduling divisible loads in distributed computing systems started in 1988 [1] and has generated considerable amount of interest among researchers and many more results are available in [2,3]. Recent results in this area are available in [4].

Divisible load scheduling problem will be more difficult, when practical issues like processor release time, finite buffer conditions and start-up time are considered. It is shown in [5] and [6], that this problem is NP hard when the buffer constraints and start-up delays in communication are included. A study on computational complexity of divisible load scheduling problem is presented

T. Srikanthan et al. (Eds.): ACSAC 2005, LNCS 3740, pp. 529–539, 2005.

in [6]. The effect of communication latencies (start-up time delays in communication) are studied in [7,8,9,10,11] using single-round and multi-round load distribution strategies. The problem of scheduling divisible loads in presence of processor release times is considered in [12,13]. In these studies, it is assumed that the processors in the network are busy with some other computation process. The time at which the processors are ready to start the computation of its load fraction (of the divisible load) is called "release time" of the processors.

The release time of processors in the network, affect the partitioning and scheduling the divisible load. In [12], scheduling divisible loads in bus network (homogeneous processors) with identical and non-identical release time are considered. The heuristic scheduling algorithms for identical and non-identical release time are derived based on multi-installment scheduling technique presented based on the release time and communication time of the complete load. In [13], heuristic strategies for identical and non-identical release time are presented for divisible load scheduling in linear network. In these studies the problem of obtaining the optimal sequence of load distribution by the root processor is not considered.

In this paper, the divisible load scheduling problem with arbitrary processor release time in single level tree network is considered. When the processors in the network have arbitrary release time, it is difficult to obtain a closed-form expression for optimal size of load fractions. Hence, for a network with arbitrary processor release times, there are two important problems: (i) For a given sequence of load distribution, how to obtain the load fractions assigned to the processors, such that the processing time of the entire processing load is a minimum, and (ii) For a given network, how to obtain the optimal sequence of load distribution.

In this paper, Problem (i), of obtaining the processing time and the load fractions assigned to the processors, for a given sequence of load distribution is solved using a real coded hybrid genetic algorithm. Problem (ii), of obtaining the optimal sequence of load distribution is a combinatorial optimization problem. For a single-level tree network with m child processors there are $m!$ sequences of load distribution are possible. The optimal sequence of load distribution is the sequence for which the processing time is a minimum. We use genetic algorithm to obtain the optimal sequence of load distribution. The genetic algorithm for obtaining the optimal sequence of load distribution uses the results of real-coded genetic algorithm used to solve Problem (i). To the best of our knowledge, this is the first attempt to solve this problem of scheduling divisible with processor release times.

2 Definitions and Problem Formulation

Consider a single-level tree network with $(m+1)$ processors as shown in Figure 1. The child processors in the network denoted as p_1, p_2, $\cdots$, p_m are connected to the root processor (p_0) via communication links l_1, l_2, $\cdots$ l_m. The divisible load originates at the root processor (p_0) and the root processor divides the

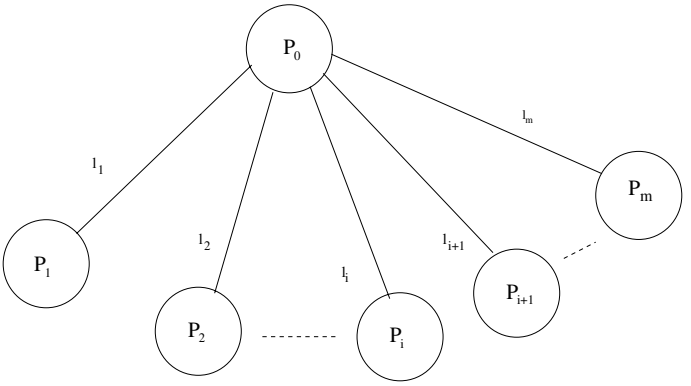


Fig. 1. Single-level tree network

load into $(m+1)$ fractions $(\alpha_0, \alpha_1, \cdots, \alpha_m)$ and keeps the part α_0 for itself to process/compute and distributes the load fractions $(\alpha_1,\alpha_2, \cdots, \alpha_m)$ to other m processors in the sequence $(p_1, p_2, \cdots, p_m)$ one after another.

The child processors in the network may not be available for computations process immediately after the load fractions are assigned to it. This will introduce a delay in starting the computation process of the load fraction. This delay is commonly referred as processor release time. The release time is different for different child processors. The objective here is to obtain the processing time and load fractions assigned to the processors. In this paper, we follow the standard notations and definitions used in divisible load scheduling literature.

Definitions:

- **Load distribution:** This is denoted as α, and is defined as an $(m+1)$-tuple $(\alpha_0, \alpha_1, \alpha_2, \cdots, \alpha_m)$ such that $0 < \alpha_i \leq 1$, and $\sum_{i=0}^{m} \alpha_i = 1$. The equation $\sum_{i=0}^{m} \alpha_i = 1$ is the normalization equation, and the space of all possible load distribution is denoted as Γ.
- **Finish time:** This is denoted as T_i and is defined as the time difference between the instant at which the i^{th} processor stops computing and the time instant at which the root processor initiates the load distribution process.
- **Processing time:** This is the time at which the entire load is processed. This is given by the maximum of the finish time of all processors; i.e.,$T = max\{T_i\}$ $i = 0, 1, \cdots, m$, where T_i is the finish time of processor p_i.
- **Release time:** The release time of a child processor p_i is the time instant at which the child processor is available to start the computation of its load fraction of the divisible load.

Notations:

- α_i: The load fraction assigned to the processor p_i.
- w_i: The ratio of the time taken by processor p_i, to compute a given load, to the time taken by a standard processor, to compute the same load;

- z_i: The ratio of the time taken by communication link l_i, to communicate a given load, to the time taken by a standard link, to communicate the load.
- T_{cp}: The time taken by a standard processor to process a unit load;
- T_{cm}: The time taken by a standard link to communicate a unit load.
- b_i: Release time for processor p_i.

Based on these notations, we can see that $\alpha_i w_i T_{cp}$ is the time to process the load fraction α_i of the total processing load by the processor p_i. In the same way, $\alpha_i z_i T_{cm}$ is the time to communicate the load fraction α_i of the total processing load over the link l_i to the processor p_i. We can see that both $\alpha_i w_i T_{cp}$ and $\alpha_i z_i T_{cm}$ are in units of time. In divisible load scheduling literature, timing diagram is the usual way of representing the load distribution process. In this diagram the communication process is shown above the time axis, the computation process is shown below the time axis and the release time is shown as shaded region below time axis. The timing diagram for the single-level tree network with arbitrary processor release time is shown in Figure 2. From timing diagram, we can write the finish time (T_0) for the root processor (p_0) as

$$T_0 = \alpha_0 w_0 T_{cp} \tag{1}$$

Now, we find the finish time for any child processor p_i. The time taken by the root processor to distribute the load fraction (α_i) to the child processor p_i is $\sum_{j=1}^{i} \alpha_j z_j T_{cm}$. Let b_i be the release time of the child processor p_i then the child processor starts the computation process at max $\left(b_i, \sum_{j=1}^{i} \alpha_j z_j T_{cm}\right)$. Hence, the

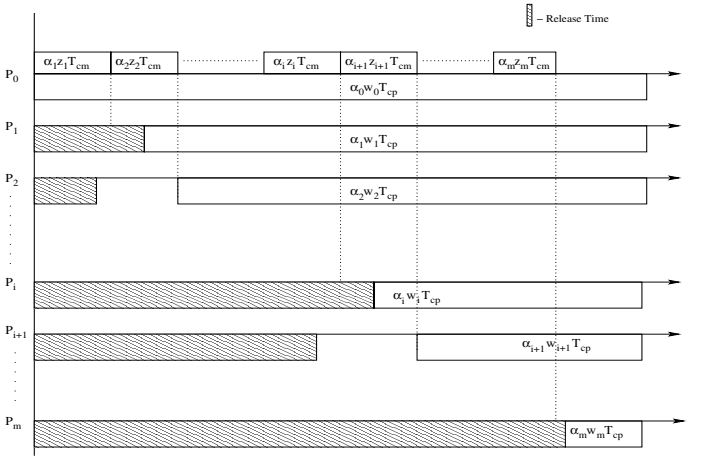


Fig. 2. Timing diagram for single level tree network with release time

finish time (T_i) of any child processor p_i is

$$T_i = \max\left(b_i, \sum_{j=1}^{i} \alpha_j z_j T_{cm}\right) + \alpha_j w_j T_{cp} \quad i = 1, 2, \cdots, m \tag{2}$$

The above equation is valid, if the load fraction assigned to the child processor is greater than zero (the child processors participate in the load distribution process) otherwise $T_i = 0$.

From the timing diagram, the processing time (T) of the entire processing load is

$$T = \max\left(T_i, \forall\ i = 0, 1, \cdots, m\right) \tag{3}$$

Obtaining the processing time (T), and the load fractions (α_i) for the divisible load scheduling problem with arbitrary processor release time is difficult. Hence, in this paper, we propose a real coded genetic algorithm approach to solve the above scheduling problem.

3 Real-Coded Genetic Algorithm

The Genetic Algorithm (GA) is perhaps the most well-known of all evolution based search techniques. The genetic algorithm is a probabilistic technique that uses a population of solutions rather than a single solution at a time [14,15,16]. In these studies, the search space solutions are coded using binary alphabet. For optimization problems floating point representation of solution in the search space outperform binary representations because they are more consistent, more precise and lead to faster convergence. This fact is discussed in [17]. Genetic algorithms using real number representation for solutions are called real-coded genetic algorithms. More details about how genetic algorithms work for a given problem can be found in literature [14,15,16,17].

A good representation scheme for solution is important in a GA and it should clearly define meaningful crossover, mutation and other problem-specific operators such that minimal computational effort is involved in these procedures. To meet these requirements, we propose real coded hybrid genetic algorithm approach to solve the divisible load scheduling problem with arbitrary processor release time. The real coded approach seems adequate when tackling problems of parameters with variables in continuous domain [18,19]. A detailed study on effect of hybrid crossovers for real coded genetic algorithm is presented in [20,21].

3.1 Real-Coded Genetic Algorithm for Divisible Load Scheduling Problem

In this paper, a hybrid real coded genetic algorithm is presented to solve the divisible scheduling problem with arbitrary processor release time. In real coded GA, solution (chromosome) is represented as an array of real numbers. The chromosome representation and genetic operators are defined such that it satisfies the normalization equation.

String Representation. The string representation is the process of encoding a solution to the problem. Each string in a population represent possible solution to the scheduling problem. For our problem, the string consists of an array of real numbers. The values of real number represents the load fractions assigned to the processors in the network. The length of the string is $m+1$, the number of processors in the system. A valid string is the one in which the total load fractions (sum of all real numbers) is equal to one.

In case of four ($m = 4$) processor system, a string will represent the load fractions $\{\alpha_0, \alpha_1, \alpha_2, \alpha_3, \alpha_4\}$ assigned to the processors p_0, p_1, p_2, p_3 and p_4 respectively. For example, a valid string $\{0.2, 0.2, 0.2, 0.2, 0.2\}$, in our problem represents the load fraction assigned to the processors ($\alpha_0 = 0.2$, $\alpha_1 = 0.2$, $\alpha_2 = 0.2$, $\alpha_3 = 0.2$ and $\alpha_4 = 0.2$) in the network. The sum of load fractions assigned to the processors is equal to 1. In general, for an m-processor system, the length of the string is equal to $m+1$.

Population Initialization. Genetic algorithms search from a population of solution points instead of a single solution point. The initial population size, and the method of population initialization will affect the rate of convergence of the solution. For our problem the initial population of solutions is selected in the following manner.

- Equal allocation: The value of load fraction (α_j) in the solution is $\frac{1}{M+1}$, for $j = 1, ..., M+1$.
- Random allocation: Generate $M+1$ random numbers. These random numbers are normalized such that the sum is equal to one. This is the value of α_j in the solution, for $j = 1, ..., M+1$.
- Zero allocation: Select a solution using equal or random allocation. Any one element in the selected solution is assigned zero and its value is equally allocated to other elements.
- Proportional allocation: The value of load fraction α_j in the solution is $\alpha_j = \frac{w_j}{\sum_{i=1}^{M} w_i}$.

Selection Function. *Normalized Geometric Ranking Method*: The solutions (population) are arranged in descending order of their fitness value. Let q be the selection probability for selecting best solution and r_j be the rank of jth solution in the partially ordered set. The probability of solution j being selected using normalized geometric ranking method is

$$s_j = q^{'} (1-q)^{r_j - 1} \tag{4}$$

where $q^{'} = \frac{q}{1-(1-q)^N}$ and N is the population size. The details of the normalized geometric ranking method can be found in [22].

Genetic Operators. Genetic operators used in genetic algorithms are analogous to those which occur in the natural world: reproduction (crossover, or recombination); mutation.

Crossover Operator: It is a primary operator in GA. The role of crossover operator is to recombine information from the two selected solutions to produce better solutions. The crossover operator improves the diversity of the solution vector. Four different crossover operators used in our divisible load scheduling problem are Two-point crossover (TPX), Simple crossover (SCX), Uniform crossover (UCX) and Averaging crossover (ACX). These crossover operators are described in [23].

Hybrid Crossover: We have used four types of crossover operators. The performance of these operators in terms of convergence to optimal solution depends on the problem. One type of crossover operator which performs well for one problem may not perform well for another problem. Hence many research works are carried out to study the effect of combining crossover operators in a genetic algorithm [24,25,26,27] for a given problem. Hybrid crossovers are a simple way of combining different crossover operators. The hybrid crossover operators use different kinds of crossover operators to produce diverse offsprings from the same parents. The hybrid crossover operator presented in this study generates eight offsprings for each pair of parents by SPX, TPX, UCX and ACX crossover operators. The most promising offsprings of the eight substitute their parents in the population.

Mutation Operator: The mutation operator alters one solution to produce a new solution. The mutation operator is needed to ensure diversity in the population, and to overcome the premature convergence and local minima problems. Mutation operators used in this study are Swap mutation (SM) and Random Zero Mutation (RZM) are described in [23].

Fitness Function. The objective in our scheduling problem is to determine the load fractions assigned to the processors such that the processing time of the entire processing load is a minimum. The calculation of fitness function is easy. The string gives the load fractions α_0, α_1, $\cdots$, α_m assigned to the processors in the network. Once the load fractions are given, the finish time of all processors can be easily obtained. For example, the finish time T_i of processor p_i is $\max\left(b_i, \sum_{j=1}^{i} \alpha_j z_j T_{cm}\right) + \alpha_i w_i T_{cp}$. If the value of α_i is zero for any processor p_i, then the finish time of that processor T_i is zero. The processing time of the entire processing load T is $\max(T_i, \forall\ i = 0, 1, \cdots, m)$. Since, the genetic algorithm maximizes the fitness function, the fitness is defined as negative of the processing time (T).

$$F = -T \tag{5}$$

Termination Criteria. In genetic algorithm, the evolution process continues until a termination criterion is satisfied. The maximum number of generations is the most widely used termination criterion and is used in our simulation studies.

3.2 Genetic Algorithm

Step 1. Select population of size N using initialization methods described earlier.
Step 2. Calculate the fitness of the solutions in the population using the equation (3).
Step 3. Select the solutions from the population using normalized geometric ranking method, for genetic operations.
Step 4. Perform different types of crossover and mutation on the selected solutions (parents). Select the N best solutions using elitist model.
Step 5. Repeat the Steps 2-4 until the termination criteria is satisfied.

4 Numerical Example

We have successfully implemented and tested the real coded hybrid genetic algorithm approach for divisible load scheduling problems in tree network with arbitrary processor release time. The convergence of the genetic algorithm depends on population size (N), selection probability (S_c), crossover rate (p_c) and mutation rate (p_m). In our simulations the numerical values used are: $N = 30$, $S_c = 0.08$, $P_c = 0.8$, $p_m = 0.2$, and T_{cm}, T_{cp} are 1.0 .

Let us consider a single-level tree network with eight child processors ($m = 8$) attached to the root processor p_0. The computation and communication speed parameters, and processor release time are given in Table 1. The root processor (p_0) distributes the load fractions to the child processors in the following sequence $p_1, p_2, \cdots, p_8$. The result obtained from real coded hybrid genetic algorithm is presented in Table 1. The processing time of the entire processing load is 0.29717

In this numerical example, the processor-link pair (p_3,l_3) is removed from the network by assigning zero load fractions. The processors p_1, p_2, p_4, p_5, p_6, p_7 and p_8 receives their load fractions at times 0.0392, 0.0815, 0.0896, 0.1397, 0.1436 0.1509 and 0.1631. Thus, the processors p_1, p_2, p_4 and p_6 will start the computation process from the release time where as the processors p_5, p_7 and p_8 will be idle until their load fractions are received.

Table 1. System Parameters and Results for Numerical Example 1

Sequence	Comp. speed parameter	Comm. speed parameter	Release Time	Load Fraction
p_0	$w_0 = 1.1$	-	-	$\alpha_0 = 0.2702$
p_1	$w_1 = 1.5$	$z_1 = 0.4$	$b_1 = 0.15$	$\alpha_1 = 0.0981$
p_2	$w_2 = 1.4$	$z_2 = 0.3$	$b_2 = 0.1$	$\alpha_2 = 0.1408$
p_3	$w_3 = 1.3$	$z_3 = 0.2$	$b_3 = 0.50$	$\alpha_3 = 0$
p_4	$w_4 = 1.2$	$z_4 = 0.1$	$b_4 = 0.2$	$\alpha_4 = 0.0810$
p_5	$w_5 = 1.1$	$z_5 = 0.35$	$b_5 = 0.05$	$\alpha_5 = 0.1431$
p_6	$w_6 = 1.2$	$z_6 = 0.1$	$b_6 = 0.25$	$\alpha_6 = 0.0393$
p_7	$w_7 = 1.0$	$z_7 = 0.05$	$b_7 = 0.1$	$\alpha_7 = 0.1462$
p_8	$w_8 = 1.65$	$z_8 = 0.15$	$b_8 = 0.12$	$\alpha_8 = 0.0812$

5 Optimal Sequence of Load Distribution

Sequence of load distribution is the order in which the child processors are activated in divisible load scheduling problem. In the earlier section, the sequence (activation order) of load distribution by the root processor is $\{p_1, p_2, \cdots, p_m\}$. For a system with m child processors there are $m!$ sequences of load distribution are possible. Optimal sequence of load distribution is the sequence of load distribution for which the processing time is a minimum.

First we will show that the problem of finding the optimal sequence of load distribution, is similar to the well-known Travelling Salesman Problem (TSP) studied in operations research literature. TSP is very easy to understand; a travelling salesman, must visit every city exactly once, in a given area and return to the starting city. The cost of travel between all cities are known. The travelling salesman to plan a sequence (or order) of visit to the cities, such that the cost of travel is a minimum. Let the number of cities be m. Any single permutation of m cities is a sequence (solution) to the problem. The number of sequences possible are $m!$. A genetic algorithm approach to TSP is well discussed in [17]. We now show the similarities between finding the optimal sequence of load distribution and Travelling Salesman Problem (TSP).

In TSP a solution is represented as string of integers representing the sequence in which the cities are visited. For example, the string {4 3 5 1 2} represents the tour as $c_4 \longrightarrow c_3 \longrightarrow c_5 \longrightarrow c_1 \longrightarrow c_2$, where c_i is the city i. In our problem, the string {4 3 5 1 2} represents the sequence of load distribution by the root processor to the child processors is $\{p_4\ p_3\ p_5\ p_1\ p_2\}$. So we can see the solution representation in our problem is the same as solution representation problem in TSP [17]. Hence, for our problem we have used the genetic algorithm approach given for TSP given in [17]. From genetic algorithm point of view, the only difference between TSP and divisible load scheduling problem is the fitness function. The fitness function is the cost of travel in TSP for the given sequence but the fitness function in our problem is the processing time for a given sequence of load distribution.

Fitness Function: The fitness function is based on the processing time of the entire processing load. Here for a given sequence, fitness function is the solution of real-coded genetic algorithm described in the earlier section. The objective in our problem is processing time minimization. Hence, in our case the fitness function is $(-T)$

$$F = -T \tag{6}$$

In order to determine the fitness (F), for any given sequence, the processing time (T), is obtained by solving the Problem (i) using real-coded genetic algorithm methodology given in the earlier section. The optimal or best sequence of load distribution can be found by using the genetic algorithm approach given for TSP given in [17], with the above fitness function.

Optimal Sequence for Numerical Example 1: The optimal sequence of load distribution (by root processor p_0) obtained using the above genetic algorithm

is $\{p_7\ p_5\ p_2\ p_8\ p_1\ p_6\ p_4\}$. The processing time for this sequence is $T = 0.2781$. The load fractions assigned to processors are: $\alpha_0 = 0.25286$, $\alpha_7 = 0.17814$, $\alpha_5 = 0.18568$, $\alpha_2 = 0.12015$, $\alpha_8 = 0.093447$, $\alpha_1 = 0.081151$, $\alpha_6 = 0.023453$, $\alpha_4 = 0.06512$. The processor p_3 is assigned a zero load fraction. Processor p_3 is assigned a zero load fraction because the release time ($b_3 = 0.5$) is high.

6 Conclusions

The problem of scheduling divisible loads in distributed computing system, in presence of processor release time is considered. In this situation, there are two important problems: (i) For a given sequence of load distribution, how to obtain the load fractions assigned to the processors, such that the processing time of the entire processing load is a minimum, and (ii) For a given network, how to obtain the optimal sequence of load distribution. A real-coded genetic algorithm is presented for the solution of Problem (i). It is shown that Problem (ii), of obtaining the optimal sequence of load distribution is a combinatorial optimization problem similar to Travelling Salesman Problem. For a single-level tree network with m child processors there are $m!$ sequences of load distribution are possible. Optimal sequence of load distribution is the sequence for which the processing time is a minimum. We use another genetic algorithm to obtain the optimal sequence of load distribution. The genetic algorithm for obtaining the optimal sequence of load distribution uses the real-coded genetic algorithm used to solve Problem (i).

Acknowledgement. This work was in part supported by MSRC-ITRC under the auspices of Ministry of Information and Communication, Korea.

References

1. Cheng, Y. C., and Robertazzi, T. G.: Distributed Computation with Communication Delay, *IEEE Trans. Aerospace and Electronic Systems*, **24**(6) (1988) 700-712.
2. Bharadwaj, V., Ghose, D., Mani, V., and Robertazzi, T. G.: *Scheduling Divisible Loads in Parallel and Distributed Systems*, Los Alamitos, California, IEEE CS Press. (1996).
3. Special Issue on: Divisible Load Scheduling, *Cluster Computing*, **6** (2003).
4. http://www.ee.sunysb.edu/tom/dlt.html
5. Drozdowski, M., and Wolniewicz, M.: On the Complexity of Divisible Job Scheduling with Limited Memory Buffers, Technical Report: R-001/2001. Institute of Computing Science, Poznan University of Technology, (2001).
6. Beaumont, O., Legrand, A., Marchal, L., and Robert, Y.: Independent and Divisible Task Scheduling on Heterogeneous Star-Shaped Platforms with Limited Memory, Research Report: 2004-22. LIP, ENS, Lyon, France. (2004).
7. Beaumont, O., Legrand, A., and Robert, Y.: Optimal Algorithms for Scheduling Divisible Work Loads on Heterogeneous Systems, *Proceedings of the International Parallel and Distributed Processing Symposium* (IPDPS'03), IEEE Computer Society Press, (2003).
8. Banini, C., Beaumont, O., Carter, L., Ferrante, J., Legrand, A., and Robert, Y.: Scheduling Strategies for Master-Slave Tasking on Heterogeneous Processor Platforms. *IEEE Trans. on Parallel and Distributed Systems*, **15**(4) (2004) 319-330.

9. Veeravalli, B., Li, X., and Ko, C. C.: On the Influence of Start-up Costs in Scheduling Divisible Loads on Bus Networks, *IEEE Trans. on Parallel and Distributed Systems*, **11**(12) (2000) 1288-1305.
10. Suresh, S., Mani, V., and Omkar, S. N.: The Effect of Start-up Delays in Scheduling Divisible Loads on Bus Networks: An Alternate Approach, *Computers and Mathematics with Applications*, **40** (2003) 1545-1557.
11. Drozdowski, M., and Wolniewicz, P.: Multi-Installment Divisible Job Processing with Communication Start-up Cost, *Foundations of Computing and Decision Sciences*, **27**(1) (2002) 43-57.
12. Bharadwaj, V., Li, H. F., and Radhakrishnan, T.: Scheduling Divisible Loads in Bus Network with Arbitary Release Time, *Computers and Mathematics with Applications*, **32**(7) (1996) 57-77.
13. Bharadwaj, V., and Wong Han Min.: Scheduling Divisible Loads on Heterogeneous Linear Daisy Chain Networks with Arbitrary Processor Release Times, *IEEE Trans. on Parallel and Distributed Systems*, **15**(3) (2004) 273-288.
14. Holland, H. J.: *Adaptation in Natural and Artificial Systems*, University of Michigan Press. Ann Arbor. (1975).
15. Goldberg, D. E.: *Genetic Algorithms in Search, Optimization and Machine Learning*, Addison-Wesley, New York. (1989).
16. David, L.: *Handbook of Genetic Algorithms*, Van Nostrand Reingold, New York. (1991).
17. Michalewicz, Z.: *Genetic Algorithms + Data Structures = Evolution Programs*, AI Series, Springer-Verlag, New York. (1994).
18. Herrera, F., Lozano, M., and Verdegay, J. L.: Tackling Real-Coded Genetic Algorithms: Operators and Tools for Behavioural Analysis, *Artificial Intelligence Review*, **12**(4) (1998) 265-319.
19. Deb, K.: *Multi-Objective Optimization using Evolutionary Algorithms*, Wiley, Chichester. (2001).
20. Herrera, F., Lozano, M., and Sanchez, A. M.: Hybrid Crossover Operators for Real-Coded Genetic Algorithms: An Experimental Study, *Soft Computing - A Fusion of Foundations, Methodologies and Applications*, (2002).
21. Herrera, F., Lozano, M., and Snchez, A. M.: A Taxonomy for the Crossover Operator for Real-Coded Genetic Algorithms: An Experimental Study, *International Journal of Intelligent Systems*, **18** (2003) 309-338.
22. Houck, C. R., Joines, J. A., and Kay, M. G.: A Genetic Algorithm for Function Optimization: A Matlab Implementation. *ACM Transactions on Mathematical Software*, **22** (1996) 1-14.
23. Mani, V., Suresh, S., and Kim, H. J.: Real-Coded Genetic Algorithms for Optimal Static Load Balancing in Distributed Computing System with Communication Delays, *Lecture Notes in Computer Science*, LNCS 3483, (2005) 269-279.
24. Herrera, F. and Lozano, M.: Gradual Distributed Real-Coded Genetic Algorithms, *IEEE Transactions on Evolutionary Computation*, **4**(1) (2000) 43-63.
25. Hong, T. P., and Wang, H. S.: Automatically Adjusting Crossover Ratios of Multiple Crossover Operators, *Journal of Information Science and Engineering*, **14**(2) (1998) 369-390.
26. Hong, T. P., Wang, H. S., Lin, W. Y., and Lee, W. Y.: Evolution of Appropriate Crossover and Mutation Operators in a Genetic Process, *Applied Intelligence*, **16** (2002) 7-17.
27. Yoon, H. S., and Moon, B. R.: An Empirical Study on the Synergy of Multiple Crossover Operators, *IEEE Transactions on Evolutionary Computation*, **6**(2) (2002) 212-223.

D3DPR: A Direct3D-Based Large-Scale Display Parallel Rendering System Architecture for Clusters

Zhen Liu, Jiaoying Shi, Haoyu Peng, and Hua Xiong

CAD&CG State Key Lab, Zhejiang University,
310027, HangZhou, PRC
{liuzhen, jyshi, phy, xionghua}@cad.zju.edu.cn

Abstract. The current trend in hardware for parallel rendering is to use clusters instead of high-end super computer. We describe a novel parallel rendering system that allows application to render to a large-scale display. Our system, called D3DPR, uses a cluster of PCs with high-performance graphics accelerators to drive an array of projectors. D3DPR consists of two types of logical nodes, Geometry Distributing Node and Geometry Rendering Node. It allows existing Direct3D9 application to run on our parallel system without any modification. The advantage of high-resolution and high-performance can be obtained in our system, especially when the triangle number of the application becomes very large. Moreover, the details of interconnecting network architecture, data distribution, communication and synchronization, etc. are hidden from the users.

1 Introduction

Modern supercomputers have allowed the scientific community to generate simulation datasets at sizes that were not feasible in previous years. The ability to efficiently visualize large, dynamic datasets on a high-resolution display is a desirable capability for researchers [5]. Regretfully, display resolution is lagging far behind. One strategy for overcoming the resolution is to tile multiple projection devices over a single display surface [3,4,8,9].

Traditionally, large-scale display environment have been driven by powerful graphics supercomputers, such as SGI's Onyx System. Unfortunately, this comes at high cost. During the past several years, high performance/cost ratio PC graphics cards have become available. So an inexpensive way to construct a large-scale display system is to use a cluster of PCs with commodity graphics accelerators to drive an array of projectors.

The design and development of systems for cluster-based large-scale display rendering is increasingly popular during the past few years. OpenGL and Direct3D are two main 3D graphics API in the world. But all of the past work focuses on the issue of developing software tools to bring sequential OpenGL applications to a large-scale display system. However, Direct3D has developed rapidly in recent

T. Srikanthan et al. (Eds.): ACSAC 2005, LNCS 3740, pp. 540–550, 2005.

years. It is widely used in multimedia, entertainment, games, 3D animation and 3D graphics computation etc. Since many Direct3D applications need to be displayed in large format, it is necessary to bring them to our large-scale display system. To accomplish this goal, we have designed and implemented D3DPR, a software system platform that make Direct3D9 applications to run efficiently on our large-scale display system, driven by PC clusters, with no modification [12].

2 Background and Related Work

2.1 Cluster Based Parallel Rendering

Most of the recent research on cluster-based rendering focuses on different algorithms to distribute the rendering of polygonal geometry across the cluster. Molnar et al. [14] classified these algorithms into three general classes based on where the sorting of primitives occurs in the transition from object to screen space. The three classes are sort-first(early during geometry processing),sort-middle(between geometry processing and rasterization) and sort-last(after rasterization).

The SGI RealityEngine [15] is a sort-middle tiled architecture that used bus-used broadcast communication to distribute primitives to a number of geometry and rasterization processors. PixelPlanes 5 [16] also provides a sort-middle tiled architecture, using a ring network to distribute primitives for a retained-mode scene description. PixelFlow [17] is a later version of PixelPlanes 5, however it is a sort-last parallel rendering system. Eldridge, Igehy and Hanrahan describe Pomegrante [18], a scalable graphics system based on point-to-point communication.

WireGL [6] is the first graphics rendering system on cluster based on sort-first architecture and is platform-independent to hardware. Chromium [7] is the successor to WireGL, a system to support OpenGL application on a cluster. Chromium provides Stream Processing Unit (SPU) and can be configured to provide a sort-first and sort-last parallel graphics architecture. AnyGL [1] is a hybrid sort-first and sort-last rendering system realized by JianYang. MSPR [2] is a retained-mode based multi-screen parallel rendering system that offers programmers OpenGL-like API.

Among those three modes, sort-first is particularly suitable for cluster implementation and can achieve scalability at near linear cost in theory.

2.2 Large-Scale Display

All systems designed to support rendering on large-scale display vary widely with respect to the way data is distributed among the cluster nodes. Chen et al. [19] first looked at the problem of data distribution. Two general models have merged, one is master-slave and the other is client-server. Furthermore, the second can be divided into mode-immediate mode and retained mode [3,11].

Large-scale display provides a large format environment for presenting high-resolution visualization by tiling together the output of a collection of projectors.

These projectors can be driven by a cluster PCs, which is augmented with high performance and low cost graphics cards.

The University of Minnesota's PowerWall [20] uses output of multiple graphics supercomputer to drive multiple projectors, creating a large tiled display for high-resolution video playback and immersive applications. The University of Illinois at Chicago extended this system to support stereo display and user tracking in the InfinityWall [21] system. The Multi-Projection system [22] developed in Princeton University adopted the idea of one node distributing OpenGL commands and other nodes rendering these primitives. OpenSG [23] provides a sort-first algorithm to use a cluster to display an image on a single display.

3 Design Issues

Because of the tremendous implemental mechanism difference between OpenGL and Direct3D, it is a very different way to implement a Direct3D-based large-scale display parallel rendering system for clusters contrast to OpenGL-based. Follows are the design issues to implement the system. In addition, our system is based on Direct3D9.

3.1 Direct3D9 Graphics Pipeline

In order to implement our Direct3D-based parallel system, let us look at the Direct3D9 graphics pipeline [13] first, as depicted in Fig. 1.

Overall the rendering pipeline is responsible for creating a 2D image given a geometric description of the 3D world and virtual camera that specifies the perspective from which the world is being viewed. The process can be divided into two parts, the geometry processing and rasterization [10]. As shown in Fig. 1, what is new in Direct3D9 is programmable pipeline. A programmable model is used for specifying the behavior of both the vertex transformation and lighting pipeline and of pixel texture blending pipeline.

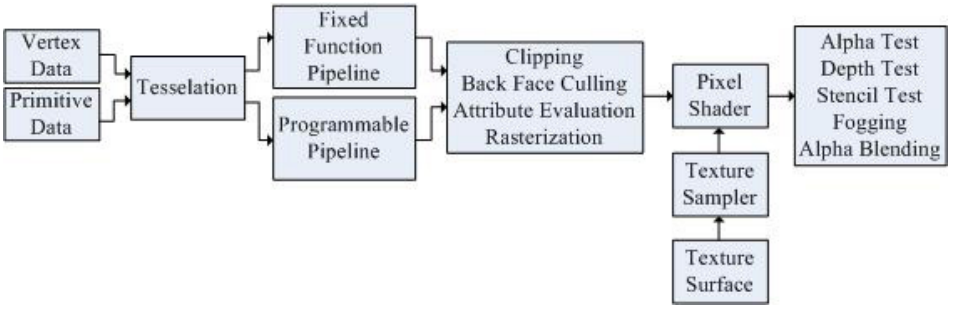


Fig. 1. Direct3D9 Rendering Pipeline

3.2 Direct3D9 Parallel Analysis

As illustrated in Fig. 1, Direct3D9 rendering pipeline has two features. First, the output of geometry transformation is the input of rasterization. The two separate processes are fit for modular implementation. Second, the basic process unit of the rendering is triangle. The computation of triangle can be processed separately. So the little data mutuality makes it easy to parallel process.

Furthermore, according to the parallel computation theory, the rendering of Direct3D9 is a kind of computation that is fit for parallel process. But it needs the distribution of computation task, collection of computation result and combination in some way. There are two kinds of methods to implement parallel algorithm. One is function parallelism, and the other is data parallelism. From the above analysis, we can see that the Direct3D9 rendering pipeline is suitable for parallel process.

3.3 Intercepting Direct3D9 COM Objects

In order to bring the Direct3D9 application to run on our system, we must obtain the rendering process of the application. Therefore we should intercept the interface method calls of the application when it is running.

We have designed a replacement Direct3D9 library to substitute the original Direct3D9 library. The interface of the new library is the same as the original one. Because Direct3D9 is based on COM, it is necessary to define classes derived from the Direct3D9 interfaces in our new library. When application invokes the Direct3D9 library to create instance of interface, the new library intercepts the interface and invokes the method to create instance of new interface, corresponding to the original interface. Finally the new interface instance will be returned. In this way, each interface can be wrapped into a new class.

3.4 Geometry Primitives Management

One important difference between Direct3D9 and OpenGL is that all geometry primitives in Direct3D9 are organized in the form of stream, i.e., vertex buffer and index buffer. Vertex\index buffer have three memory storage modes, such as system memory, AGP memory and video memory. If vertex\index buffer were stored in video memory, the time that primitives transmit from system memory to video memory could be saved. Thus the rendering speed can be improved largely.

In the new library, we allocate memory to store the vertex buffer\index buffer stream and texture stream obtained from the Direct3D9 application. In order to transmit the stream, we must record some extra information about the stream. Both vertex\index and texture have the property of format, type and size. In addition, texture has width and height property. One important thing is that we should compute the size of texture according to the format of it. Especially when the texture is compressed, the size should be computed with the following formula:

$$\max(1, width4) * max(1, height4) * 8(DXT1) or 16(DXT2 - 5). \quad (1)$$

width refers to the width of the surface of texture, height refers to the height of the surface of the texture. DX1-5 refers to compression texture format [13].

4 D3DPR System Architecture

We designed D3DPR, a Direct3D-based large-scale display parallel rendering system for clusters. D3DPR is organized as sort-first parallel rendering system. It consists of two logical nodes, Geometry Distributing Node (G-node) and Geometry Rendering Node (R-node), as shown in Fig. 2. Following we will illustrate the implementation of the system architecture in details.

4.1 Geometry Distributing Node Implementation

The D3DPR library is implemented as a complete replacement of the system Direct3D9 library, no commands are added for users. The task of D3DPR library has three aspects. First, it intercepts and encodes the COM object of the application. Second, it packs the parameters of interface method call. Third, it retrieves vertex\ index buffer and texture stream of the application. D3DPR_Client can get rendering information from D3DPR library and broadcast them to the D3DPR_Server of R-node for rendering.

In D3DPR, Direct3D9 interface methods have been divided into four categories, 3D environment initialization command, rendering model command, rendering command and special command.

3D environment initialization command: When rendering with Direct3D9, an application must perform a series of tasks to select an appropriate Direct3D9 device. It can query hardware to detect the supported Microsoft Direct3D9 device types, including enumerating display adapters and selecting Direct3D9 devices. This kind of interface method can be classified as the 3D environment

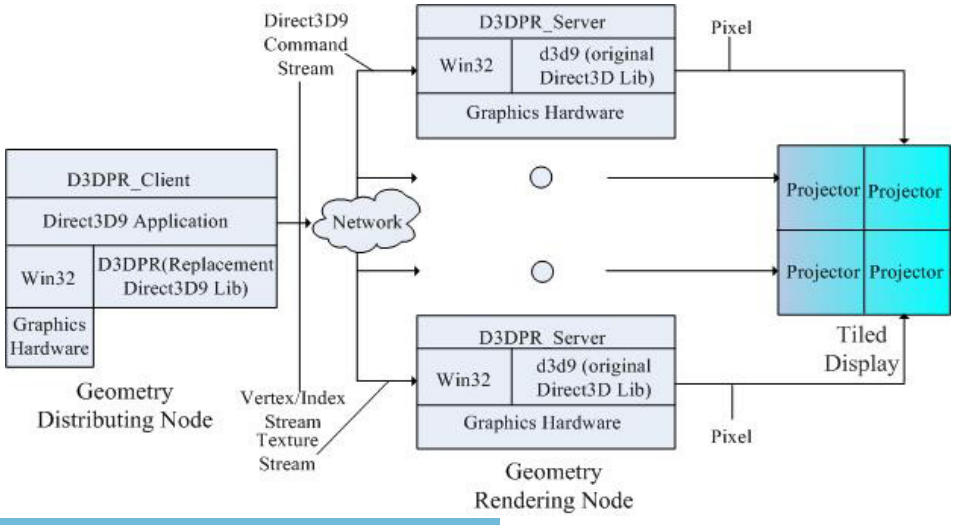


Fig. 2. Direct3D9 Rendering Pipeline

initialization command, such as IDirect3D9:: EnumAdapterModes, IDirect3D9:: CheckDeviceType, IDirect3D9:: GetDeviceCaps, IDirect3D9:: GetAdapterDisplayMode etc.

Rendering model command: In general, the rendering model can be represented by vertex buffer\index buffer stream and texture stream. Vertex buffers, represented by the IDirect3DVertexBuffer9 interface, are memory buffers that contain vertex data. Index Buffers, represented by the IDirect3DIndex Buffer9 interface, are memory buffers that contain index data. Texture is a bitmap of pixel colors that give an object realistic visual complexity, represented by IDirect3DBaseTexture9, IDirect3DTexture9 and IDirect3DvolumeTexture9. The interface methods that can get the entire rendering mode, including all the above streams, belong to this kind of command.

Rendering command: Rendering command is directly responsible for rendering process, such as setting transformation matrix, set material, set lighting, set texture, draw primitive etc. Most of methods in IDirect3DDevice9 interface can be classified to this category.

Special command:IDirect3DDevice9::Clear clears the viewport, or a set of rectangles in the viewport, to a specified RGBA color, clears the depth buffer, and erases the stencil buffer. IDirect3DDevice9::Present presents the contents of the next buffer in the sequence of back buffers owned by the device. The two interface methods belong to special command.

We allocate three memory buffers in G-node. They are Direct3D9 command stream buffer, vertex\index buffer stream buffer and texture stream buffer. When D3DPR meets interface method call in the process of intercepting the Direct3D9 application, the method is packed immediately into the Direct3D9 command stream buffer and waits for buffer-flush. As soon as the buffer is filled up, the G-Node will broadcast the buffer to all the R-Nodes. However, the retrieved vertex buffer\index buffer stream and texture stream are packed into corresponding buffer and broadcasted to the R-node immediately.

4.2 Geometry Rendering Node Implementation

When one R-node receives package from G-node, it first determines which kind of stream that it belongs to. Four kinds of Direct3D9 commands are executed in their different ways.

For 3D environment initialization command, R-node calls the corresponding Direct3D9 hardware directly after decoding. One important thing of it is the reconstruction of COM object according to its coding.

For rendering model command, R-node needs to reconstruct the vertex buffer\ index buffer data and texture data. It is usually occur before the rendering process. Because each R-node has reserved one copy of the data, R-node can render directly without receiving data from G-node every frame. Therefore, the render speed can be increased greatly contrast to the OpenGL-based immediate-mode parallel rendering system.

For special methods, R-node execute only once for each frame. IDirect3DDevice9::Present will not be executed until all previously received interface methods have been executed. To synchronize IDirect3DDevice9:: Present method for all R-nodes, a global barrier operation must be executed before calling hardware IDirect3DDevice9::Present.

Because each R-node is responsible for rendering one project, it draws only a sub- frustum of the original scene. The output of each R-node is connected to a projector that projects onto a common screen. These projectors are configured into a tiled array so that their outputs produce a single large image.

4.3 Network Transmission

There are several message-passing libraries for programming of parallel environments, such as PVM and MPI standard [24,25]. We adopt MPI as our network transmission tool. Even though MPI is designed as communication API for multiprocessor computers, it can also work on cluster of PCs. In MPI, there are two different communication methods. Group members can either communicate pairwise, or they can communicate with all members of the group. The first one is called point-to-point communication, and the second is called collective communication. In our system, we define the G-node and all the R-nodes as a group. So the G-node can broadcast the command stream, vertex\index buffer stream and texture stream to all the R-nodes by use of MPI_Bcast( ). And all the R-nodes can receive them by MPI_Bcast( ), too.

4.4 Synchronization

In our system, The G-node includes D3DPR_Client process and Direct3D9 application process. So the synchronization contains two aspects, one is the between application and D3DPR_Client, and the other is between D3DPR_Client and D3DPR_Server.

We adopt the shared memory method to realize the synchronization between the application process and D3DPR_Client process. In addition, we make use of MPI_Barrier( ) to ensure the synchronization between the D3DPR_Client and D3DPR_Server. It blocks calling process until all members of the group associated with communicator are blocked at this barrier. In the implementation of our system, we not only ensure the correct stream sending and receiving between D3DPR_Client and D3DPR_Server, but also block the D3DPR_Server process before they present the contents of the next buffer in the sequence of back buffers owned by the device.

5 Experimental Results

D3DPR is a Direct3D-based large-scale parallel rendering system, thus it inherited both advantage of high-resolution from large-scale display and high-performance from parallel rendering. We analyze the two aspects of D3DPR in the following.

5.1 Experimental Environment and Benchmark Application

Our experiment is performed on five PCs connected via 1000M Ethernet. Each PC has Supermicro X5DAL-G mainboard supporting AGP8X graphics card data transmission, two Intel XEON 2.4GHz CPU, 1GB Kingston DDR memory, NVIDIA Geforce FX5950 graphics card and Intel Pro 1000MT dual ports network card. We execute the following application with one PC as G-node and four PCs as R-node. And each R-node PC drives one TOSHIBA TLP-T620 LCD Projector.

We selected two representative applications for experiment and analysis. The both applications are the samples provided by Microsoft Direct3D9 SDK.

Tiger: It is a mesh sample that shows how to load and render file-base geometry meshes in Direct3D9.

Billboard: It illustrates the billboarding technique. Rather than rendering complex 3-D models, such as a high-polygon tree model, billboarding renders a 2-D image of the model and rotates it to always face the eyepoint.

5.2 Results

Table 1 shows the results we have gathered from running the benchmark applications on our system. The results show that when the triangles number of the application is relatively smaller, its frame rate decreased after it has been distributed. This circumstance is ascribed to the reason that the overhead of network has overrun the advantage of parallel rendering. But with the triangles number increasing, the speed of parallel rendering becomes higher than that of single PC.

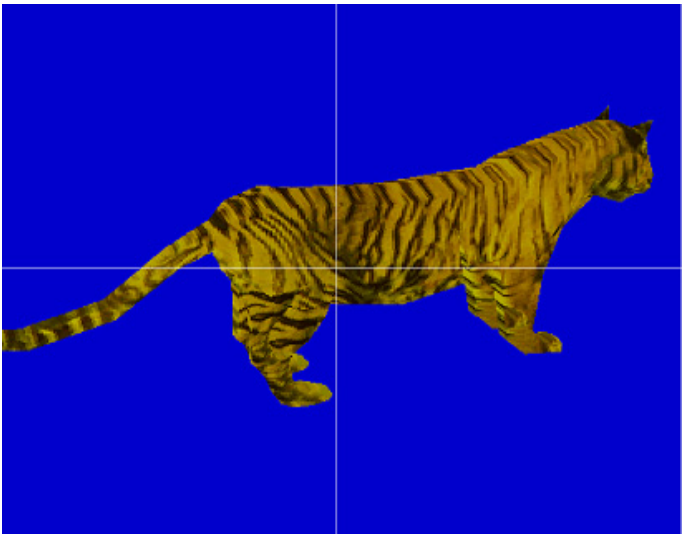


Fig. 3. High-Resolution Tiger Snapshots

Table 1. Performance of Two Benchmarking Applications

Benchmark Application	Triangles /Frame	Frame Rate FPS(1PC)	Frame Rate FPS(2PC)	Frame Rate FPS(4PC)	Speedup (2PC)	Speedup (4PC)
Tiger	599	256	185	164	0.72	0.64
Billboard	3,842	112	118	128	1.05	1.14

Fig. 3 and Fig. 4 shows the main advantage of our D3DPR system: high-resolution. They are respectively one snapshot of our two benchmark application running on our four R-nodes, and the resolution of each picture is 2048*1536.

Fig. 4. High-Resolution Billboard Snapshots

6 Conclusions and Future Work

We have described D3DPR, a Direct3D-based large-scale display parallel rendering system architecture for clusters. Since it integrates the advantages of both large-scale display and cluster based parallel rendering, it can offer both high-resolution and high-performance to the users. Provided with Direct3D9 interface, existing application written with Direct3D9 can be run directly on our system without any modification.

Direct3D is different from OpenGL since it is built on COM. We have realized interpreting Direct3D9 COM object on G-node and reconstructing them on R-node. In addition, all the parameters of each interface method can be packed and unpacked correctly.

We have adopted client-server mode to distribute data among the nodes. Because all geometry primitives in Direct3D9 are organized in the form of stream, such as vertex buffer and index buffer, the primitives don't need to transmit every frame. Thus it can improve rendering speed largely. Moreover, we can support the retrieval, transmission and reconstruction the texture data.

There is much work we can do to improve the performance of our system in the future, such as stream-based bounding box computation and rendering state tracking. Furthermore, we will support GPU program on our parallel rendering system.

Acknowledgements

This paper is partially supported by National Basic Research Program of China (973 Program) No.2002CB312100.

References

1. Jian Yang, Jiaoying Shi, Zhefan Jin, and Hui Zhang.: Design and Implementation of A Large-scale Hybrid Distributed Graphics System. Eurographics Workshop on Parallel Graphics and Visualization, Saarbruecken, Germany, 2002
2. Chao Li, Zhefan Jin, and Jiaoying Shi.: MSPR:A Retained-Mode Based Multi-screen Parallel Rendering System. The the 4th International Conference on Virtual Reality and its Application in Industry, Tianjin, P.R.China, 2003
3. Oliver G.Staadt, Justin Walker, Christof Nuber, and Bernd Hamann.: A Survey and Performance Analysis of Software Platforms for Interactive Cluster-Based Multi-Screen Rendering. 7th International Workshop on Immersive Projection Technology, 9 the Eurographics Workshop on Virtual Enviroments, Zurich, Switzerland, 2003
4. Greg Humphreys, Pat Hanrahan.: A Distributed Graphics System for Large Tiled Displays. IEEE Visualization '99, pages 215–224, October 1999.
5. Greg Humphreys, Ian Buck, Matthew Eldridge, and Pat Hanrahan.: Distributed Rendering for Scalable Displays. Proceedings of Supercomputing 2000
6. Greg Humphreys, Matthew Eldridge, Ian Buck, Gordon Stoll, Matthew, and Pat Hanrahan.: WireGL: A Scalable Graphics System for Clusters. In Proceedings of ACM SIGGRAPH 2001
7. Greg Humphreys, Mike Houston, Ren Ng, Randall Frank, Sean Ahern, and Peter D.Kirchner.: Chromium: A Stream-Processing Framework for Interactive Rendering on Clusters. ACM Transactions on Graphics, vol 21(3), pp. 693-702, Proceedings of ACM SIGGRAPH 2002
8. Yuqun Chen, Han Chen, Douglas W. Clark, Zhiyan Liu, Grant Wallace, and Kai Li.: Software Environments for Cluster-based Display Systems (2001). The First IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid 2001), Brisbane, Australia, 15-18 May 2001.
9. Michael S.Brown, W.Brent Seales.: A Practical and Flexible Tiled Display System. Proceeding of the 10 th Pacific Conference on Computer Graphics and Applications, 2002

10. Dirk Bartz, Bengt-Olaf Schneider, and Claudio Silva.: Rendering and Visualization in Parallel Environments. Course on Rendering and Visualization in Parallel Environments of ACM SIGGRAPH 1999
11. T.van der Schaaf, L. Renambot, D.Germans, H. Spoelder, and H.Whitlock.: Retained Mode Parallel Rendering for Scalable Tiled Displays. Immersive Projection Technologies Symposium, 2002
12. David Gotz.: Design Considerations for a Multi-Projector Display Rendering Cluster. University of North Carolina at Chapel Hill Department of Computer Science Integrative Paper, 2001
13. Microsoft DirectX 9.0 Update (Summer 2003). http://www.microsoft.com/ downloads.
14. S.Molnar, M.Cox, D.Ellsworth, and H.Fuchs.: A Sorting Classification of Parallel Rendering. IEEE Computer Graphics and Applications, 23-32,1994
15. K Akeley.: Realityengine graphics. Computer Graphics Proceedings, Annual Conference Series, ACM SIGGRAPH, Anaheim, CA, 1993:109 116
16. H. Fuchs, J. Poulton, et al.: Pixel Plane 5: A Heterogeneous Multiprocessor Graphics System using Processor Enhanced Memories. Computer Graphics Proceedings, Vol.23, No.3, July 1989, pp.79-88
17. J. Eyles, S. Molnar, J. Poulton, T. Greer, A. Lastra, N. En-gland, and L. Westover.: PixelFlow: The Realization. Proceedings of the 1997 Siggraph/Eurographics Workshop on Graphics Hardware, Los Angles, CA, Aug.3-4, 1997, pp.57-68
18. M Eldridge, H Igehy, and P Hanrahan.: Pomegranate: a fully scalable graphics architecture. Computer Graphics Proceedings, Annual Conference Series, ACM SIGGRAPH, New Orleans, Louisiana, 2000:443 454
19. H. Chen, Y. Chen, A. Finkelstein, T. Funkhouser, K. Li, Z. Liu, R. Samanta, and G. Wallace.: Data Distribution Strategies for High Resolution Displays. Computer and Graphics Vol.25, 811-818,2001
20. University of Minnesota," PowerWall homepage." http://www.lcse.umn.edu/ research/powerwall/powerwall.html
21. M. Czernuszenko, D. Pape, D. Sandin, T. DeFanti, G. Dawe, and M.Brown.: The ImmersaDesk and Infinity Wall Projection-Based Virtual Reality Displays. In Computer Graphics, May 1997
22. Rudrajit Samanta, Jiannan Zheng, Thomas Funkhouser, Kai Li, and Jaswinder Pal Singh.: Load Balancing for Multi-Projector Rendering Systems. In Proceedings of the SIGGRAPH/ Eurographics Workshop on Graphics Hardware, Aug, 1999
23. G.Voss, J. Behr, D.Reiners, and M.Roth.: A Multi-thread Safe Foundation for Scene Graphs and its Extension to Clusters. Eurographics Workshop on Parallel Graphics and Visualization, 33-38,2002
24. The Message Passing Interface (MPI) standard. http://www-unix.mcs.anl.gov/ mpi/
25. Message Passing Interface Forum. http://www.mpi-forum.org/

Determining Optimal Grain Size for Efficient Vector Processing on SIMD Image Processing Architectures*

Jongmyon Kim[1], D. Scott Wills[2], and Linda M. Wills[2]

[1] Chip Solution Center, Samsung Advanced Institute of Technology,
San 14-1, Nongseo-ri, Kiheung-eup, Kyungki-do, 449-712, South Korea
jongmyon.kim@samsung.com
[2] School of Electrical and Computer Engineering,
Georgia Institute of Technology, Atlanta, Georgia 30332-0250
{scott.wills, linda.wills}@ece.gatech.edu

Abstract. Adaptable silicon area usage within an integrated pixel processing array is a key issue for embedded single instruction, multiple data (SIMD) image processing architectures due to limited chip resources and varying application requirements. In this regard, this paper explores the effects of varying the number of vector (multichannel) pixels mapped to each processing element (VPPE) within a SIMD architecture. The VPPE ratio has a significant impact on the overall area and energy efficiency of the computational array. Moreover, this paper evaluates the impact of our color-aware instruction set (CAX) on each VPPE configuration to identify ideal grain size for a given SIMD system extended with CAX. CAX supports parallel operations on two-packed 16-bit (6:5:5) YCbCr (luminance-chrominance) data in a 32-bit datapath processor, providing greater concurrency and efficiency for vector processing of color image sequences. Experimental results for 3-D vector quantization indicate that high processing performance with the lowest cost is achieved at VPPE = 16 with CAX.

1 Introduction

An important issue for embedded SIMD image processing architectures is determining the ideal grain size that provides sufficient processing performance with the lowest cost and the longest battery life for target applications [7, 11]. In color imaging applications, the grain size of the processing elements (PEs) determines the number of vector (multicannel) pixels that are mapped to each PE, which is called the vector-pixel-per-processing-elements (VPPE) ratio. The VPPE ratio has a significant impact on the overall area and energy efficiency of the computational array. To explore the effects of different VPPE ratios on performance and efficiency for a specified SIMD architecture and implementation technology, a cycle-accurate SIMD simulator and a technology modeling tool are used. The SIMD simulator supports application development and execution evaluation including cycle counts, dynamic instruction frequencies, and system utilization. The technology modeling tool

* This work was performed by authors at the Georgia Institute of Technology (Atlanta, GA).

T. Srikanthan et al. (Eds.): ACSAC 2005, LNCS 3740, pp. 551 – 565, 2005.

estimates technology parameters such as system area, power, latency, and clock frequency. These application and technology parameters are combined to determine execution time, area efficiency, and energy consumption for each VPPE mapping.

Moreover, this paper evaluates and contrasts our color-aware instruction set (CAX) and MDMX [17] (a representative MIPS multimedia extension) for different VPPE configurations to identify the most efficient architecture that delivers the required computational power of the workload while achieving the highest area and energy efficiency. Unlike typical multimedia extensions (e.g., MMX [20], VIS [23], and MDMX), CAX exploits parallelism within the human perceptual color space (e.g., YCbCr). Rather than depending solely on generic subword parallelism [16], CAX supports parallel operations on two-packed 16-bit (6:5:5) YCbCr data in a 32-bit datapath processor. The YCbCr space allows coding schemes that exploit the properties of human vision by truncating some of the less important data in every color pixel and allocating fewer bits to the high-frequency chrominance components that are perceptually less significant. Thus, the compact 16-bit color representation consisting of a 6-bit luminance (Y) and two 5-bit chrominance (Cb and Cr) components provides satisfactory image quality [13, 14]. For a performance and efficiency comparison, MDMX (e.g., operating four 8-bit pixels in a 32-bit register) was chosen as a basis of comparison because it provides an effective way of dealing with reduction operations, using a wide packed accumulator that successively accumulates the results produced by operations on multimedia vector registers. Other multimedia extensions provide more limited support of vector processing in a 32-bit datapath processor without accumulators.

Experimental results for 3-D vector quantization using application simulation and technology modeling indicate that as the VPPE ratio decreases, the sustained throughput increases because of more data parallelism (or an increase in available PEs). However, the most effective VPPE ratio does not occur at VPPE = 1 in the combination of performance and efficiency. The most efficient operation for the task is achieved at VPPE = 16 with CAX, delivering in excess of 280 gigaoperations per second using 280 mm^2 of silicon area and 4.2W of power for 4,096 PEs running at 50 MHz clock frequency and 100 nm technology. Moreover, CAX outperforms MDMX for all the VPPE configurations in terms of processing performance, area efficiency, and energy efficiency due to greater subword parallelism and reduced pixel word storage.

The rest of this paper is organized as follows. Section 2 discusses related research. Section 3 presents an overview of color image processing and the selected vector quantization. Section 4 describes VPPE ratio, the design variable in this study. Section 5 presents modeled SIMD architectures and a summary of the color-aware instruction set. Section 6 illustrates a simulation infrastructure, and Section 7 evaluates the system area and power for our modeled architectures. Section 8 analyzes execution performance and efficiency for each case. Section 9 concludes this paper.

2 Related Research

In the last decade, with the rapid progress in VLSI technology, tremendous numbers of transistors have enabled the monolithic integration of traditional imaging systems such as a charge-coupled device (CCD) array, an analog-to-digital conversion (ADC)

unit, and a DSP [6]. The performance of these systems, however, is limited by serialized communications between the different modules. As a solution, CMOS image sensors allow direct pixel access and enable their ability to be co-located [5] or vertically integrated [2] with the CMOS computing layer. However, none of these systems have addressed the issue of how much processing capability is needed for each PE per pixel directly mapped to it.

Recently, Gentile et al. have explored the impact of varying granularity of mapping an image to the PE array [7]. In [11], Herbordt et al. examined the effects of varying the array size, the datapath, and the memory hierarchy on both cost and performance. However, both of these studies measured processing performance and efficiency on sets of grayscale (1-D) image processing applications, failing to provide a quantitative understanding of performance and efficiency with respect to vector (multichannel) processing for different PE configurations.

This paper evaluates the effects of different VPPE ratios with respect to vector processing for a specified SIMD architecture and implementation technology. This paper also evaluates the impact of CAX on each VPPE configuration to identify the most efficient PE granularity.

3 3-D Vector Quantization

In multichannel picture coding, standard images represent vector-valued image signals in which each pixel can be considered to be a vector of three components (e.g., red, green, and blue). However, the RGB space is ill-suited for the human perception of color. As a result, applying image processing techniques in the RGB space often produces color distortion and artifacts [24]. In addition, each R, G, and B component is highly correlated and thus not well-suited for independent coding. To overcome these problems, the image and video processing community widely uses the YCbCr space, a color coordinate space based on the human perception of color. Since the human visual system is less sensitive to high frequencies in chrominance (Cb and Cr), chrominance components can be subsampled without a perceivable distortion of color [9]. Further, the YCbCr space allows luminance processing independent chrominance components. Because of these properties, both separate channel processing and luminance only processing are used in color imaging applications, yielding usable results [4, 10, 15]. However, both of these approaches may not be able to extract certain crucial information conveyed by color because they do not account for the correlation between color channels. This results in reducing the accuracy of the process and the overall image quality. A proper vector approach to color manipulation would be much more beneficial [21]. Thus, this paper considers the problem of color image processing in vector space.

Full search vector quantization (FSVQ), a promising candidate for low rate and low power image and video compression, was selected for a case study. It has a computationally inexpensive decoding process and low hardware requirement for decompression, while still achieving an acceptable picture quality at high compression ratios [8]. However, the encoding process is computationally very intensive. Computational cost can be reduced by using suboptimal approaches such as tree-searched vector quantization (TSVQ) [8]. This study prefers to overcome the

computational burden by using a parallel implementation of FSVQ on a SIMD array system. Figure 1 illustrates how FSVQ is performed in image compression systems. FSVQ is defined as the mapping of k-dimensional vectors in the vector space $\mathbf{R}^k$ into a finite set of vectors $\mathbf{V} = \{\mathbf{c}_i, i=1,\ldots,N\}$, where N is size of the codebook. Each vector is called as a code vector or codeword. Only index i of the resulting code vector is sent to the decoder. At the decoder, an identical copy of the codebook is looked up by a simple table-lookup operation.

In the experiment, each input image (e.g., 256 × 256 pixels) is subdivided into macro blocks of 4 × 4 pixels or vectors of 16 elements (i.e., k = 16). Each input vector is then compared with template patterns in the codebook to find the best match in terms of the chosen cost function. In the 2-D case, non-overlapping vectors are extracted from the input image by grouping a number of contiguous pixels to retain available spatial correlation of data. The input blocks are then compared with the codebook in a parallel systolic fashion, with a large number of them compared at any given time in parallel. A key enabling role is played by the toridal structure of the interconnection network, which enables communication among the nodes on opposite edges of the mesh. Seven different PE configurations having different VPPEs and local memory are evaluated to identify the most efficient operation for FSVQ.

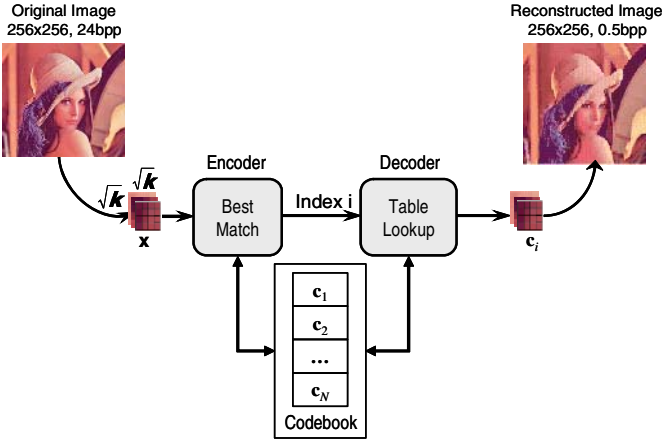


Fig. 1. A vector quantization system for color image compression

4 VPPE Ratio

The VPPE ratio, which is defined as the number of vector pixels mapped to each processor within a SIMD architecture, is selected as the design variable in this study. Figure 2 pictorially illustrates the assignment of vector pixels based on the VPPE ratio. In this study, seven different VPPE values are used, defined as $\text{VPPE} = 2^{2i}$, i = 0,…,6. The corresponding number of processing elements is defined as $N_{PE} = N_{img}/\text{VPPE}$ in which N_{img} is the number of pixels in the image. Since all the configurations use a fixed three-band 256 × 256 pixel image, the number of PEs in a 256 × 256 pixel system is determined to be $N_{PE} = 4^{(8-i)}$, i = 0,…,6. Modeled SIMD configurations and their parameters are described next.

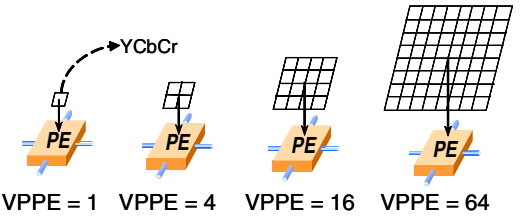


Fig. 2. Examples of vector pixels per processing element ratio

5 Modeled SIMD Architectures

5.1 SIMD Configurations and Local Memory Sizes

A data parallel SIMD architecture shown in Figure 3 is studied as a baseline PE for this study. This SIMD architecture is symmetric, having an array control unit (ACU) and an array consisting of from a few ten to a few thousand PEs. Depending on the VPPE ratio, each SIMD configuration has a different VPPEs and a different local memory to store input images and temporary data produced during processing. With sensor sub-arrays, each PE is associated with a specific portion of an image frame, allowing streaming pixel data to be retrieved and processed locally. Each PE has a reduced instruction set computer (RISC) datapath and familiar functional units (e.g., ALU and MACC).

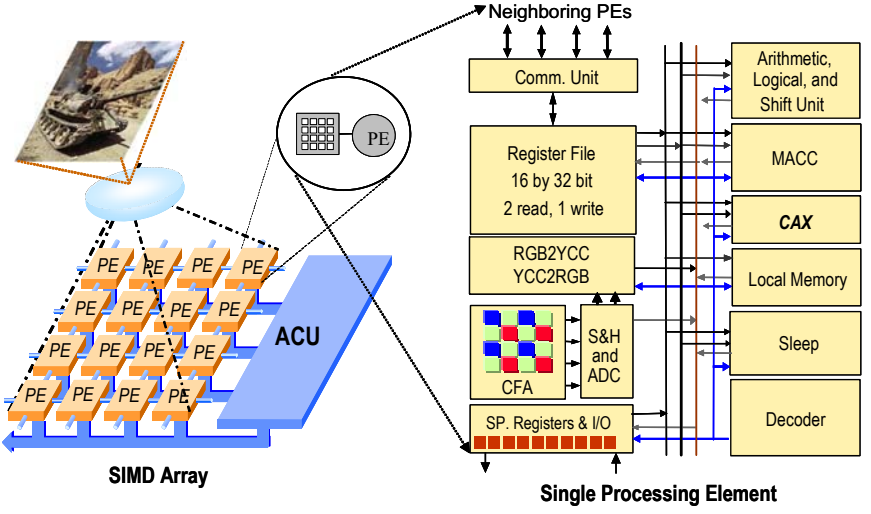


Fig. 3. Block diagram of a SIMD array and a processing element

Moreover, CAX instructions (see Section 5.2) are included in the instruction set architecture (ISA) of the SIMD array to improve the performance of color imaging applications. For a performance comparison, MDMX-type instructions are also included in the SIMD ISA. In the experiment, the overhead of the color conversion

was not included in the performance evaluation for all the implementations. In other words, this study assumes that the baseline, MDMX, and CAX versions of the task directly support YCbCr data in the band-interleaved format (e.g., four-packed 8 bit |Unused|Cr|Cb|Y| for baseline and MDMX and two-packed 6-5-5 bit |Cr|Cb|Y|Cr|Cb|Y| for CAX). Moreover, since each CAX configuration requires smaller pixel word storage than the corresponding baseline and MDMX configurations, the local memory size is set to twice the VPPE ratio for the CAX configurations but four times the VPPE ratio for the baseline and MDMX configurations, except for VPPE = 1 where eight words are used for all three versions. Table 1 describes all the SIMD configurations and their local memory sizes. An overview of the CAX instructions is presented next.

Table 1. Modeled SIMD configurations and their parameters

Parameter	Value						
Number of PEs	65,536	16,384	4,096	1,024	256	64	16
VPPEs	1	4	16	64	256	1,024	4,096
Base (Memory/PE)	8	16	64	256	1,024	4,096	16,384
MDMX (Memory/PE)	8	16	64	256	1,024	4,096	16,384
CAX (Memory/PE)	8	8	32	128	512	2,048	8,192
VLSI Technology	100 nm						
Clock Frequency	50 MHz						
Interconnection Network	Mesh						
intALU/intMUL/Barrel Shifter/intMACC/Comm	1 / 1 / 1 / 1 / 1						

5.2 Color-Aware Instruction Set

The color-aware instruction set (CAX) is a set of instructions targeted at accelerating vector processing of color image sequences. CAX, shown in Figure 4, supports parallel operations on two-packed 16-bit (6:5:5) YCbCr data in a 32-bit datapath processor.

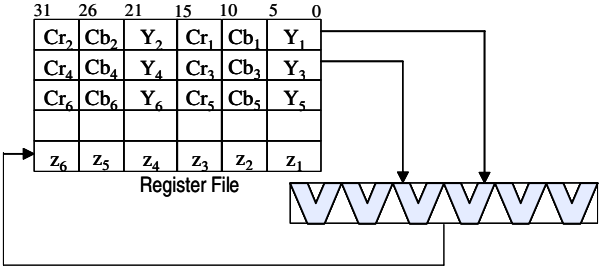


Fig. 4. An example of a partitioned ALU functional unit of CAX

In addition, CAX employs a 128-bit color-packed accumulator that provides a solution to overflow and other issues caused by packing data as tightly as possible by implicit width promotion and adequate space. The CAX instructions are classified into four different groups (see [6] for an in-depth description of CAX):

Parallel arithmetic and logical instructions. The CAX arithmetic and logical instructions include `ADD_CRCBY` (signed, unsigned saturation, and modulo), `SUBTRACT_CRCBY` (signed, unsigned saturation, and modulo), `MULTIPLY_CRCBY`, and `AVERAGE_CRCBY`, the most frequent operations in many image and video computations. The `AVERAGE_CRCBY` instruction, for example, is useful for performing motion compensation [4] in which a new image frame is created by averaging two different image blocks.

Parallel compare instructions. The CAX compare instructions include `CMPEQ(N)_CRCBY`, `CMPGT_CRCBY`, `CMOV_CRCBY` (conditional move), `MIN_CRCBY`, and `MAX_CRCBY`. These instructions compare pairs of the sub-elements (e.g., Y, Cb, and Cr) in the two source registers. The first three instructions, for example, are useful for a condition query performed on the incoming data such as a chroma-keying algorithm [18] in which the `CMPEQN_CRCBY` instruction builds a mask that is a sequence of 1's for true results and 0's for false results. The `CMOV_CRCBY` instruction then uses the mask for identifying which pixels to keep from two input images while building a final picture. The latter two instructions, `MIN_CRCBY` and `MAX_CRCBY`, are especially useful for performing a median filter in which these instructions accelerate a bubble-sort algorithm by comparing pairs of the sub-elements in the two source registers in parallel while outputting the minimum and maximum values to the target register.

Permute instructions. The permute CAX instructions include `MIX_CRCBY` and `ROTATE_CRCBY`. These instructions are used to reorder the sub-elements in a register. The mix instruction mixes the sub-elements of the two source registers into the operands of the target register and the rotate instruction rotates the sub-elements to the right by an immediate value. These instructions are useful for performing a matrix transposition [22].

Special-purpose instructions. Special-purpose CAX instructions include `ADACC_CRCBY` (absolute-differences-accumulate), `MACC_CRCBY` (multiply accumulate), `RAC` (read accumulator), and `ZACC` (zero accumulator), which provide the most computational benefits of all the CAX instructions. The `ADACC_CRCBY` instruction, for example, is frequently used in a number of block-matching algorithms (BMAs). The `MACC_CRCBY` instruction is useful in DSP algorithms that involve for computing a vector dot-product, such as digital filters and convolutions. The latter two instructions `RAC` and `ZACC` are related to the managing of the CAX accumulator.

6 Simulation Infrastructure

Figure 5 shows an overview of the simulation infrastructure which is divided into three levels: application, architecture, and technology. At the application level, an instruction-level SIMD simulator has been used to profile execution statistics, such as cycle counts,

dynamic instruction frequencies, and system utilization, for the three different versions of the task: (1) baseline ISA without subword parallelism (base-SIMD), (2) baseline plus MDMX ISA (MDMX-SIMD), and (3) baseline plus CAX ISA (CAX-SIMD). At the architecture level, the heterogeneous architectural modeling (HAM) of functional units for SIMD arrays [21] has been used to calculate the design parameters of the modeled architectures. For the design parameters of the MDMX and CAX functional units (FUs), Verilog models for the baseline, MDMX, and CAX FUs were implemented and synthesized with the Synopsys design compiler (DC) using a 0.18-micron standard cell library. The reported area specifications of the MDMX and CAX FUs were then normalized to the baseline FU, and the normalized numbers were applied to the HAM tool for determining the design parameters of MDMX- and CAX-SIMD. The design parameters are then passed to the technology level. At the technology level, the Generic System Simulator (GENESYS) developed at Georgia Tech [19] has been used to calculate technology parameters (e.g., latency, area, power, and clock frequency) for each configuration. Finally, the databases (e.g., cycle times, instruction latencies, instruction counts, area, and power of the functional units), obtained from the application, architecture, and technology levels, were combined to determine execution times, area efficiency, and energy efficiency for each case. The next section evaluates the system area and power of the modeled architectures.

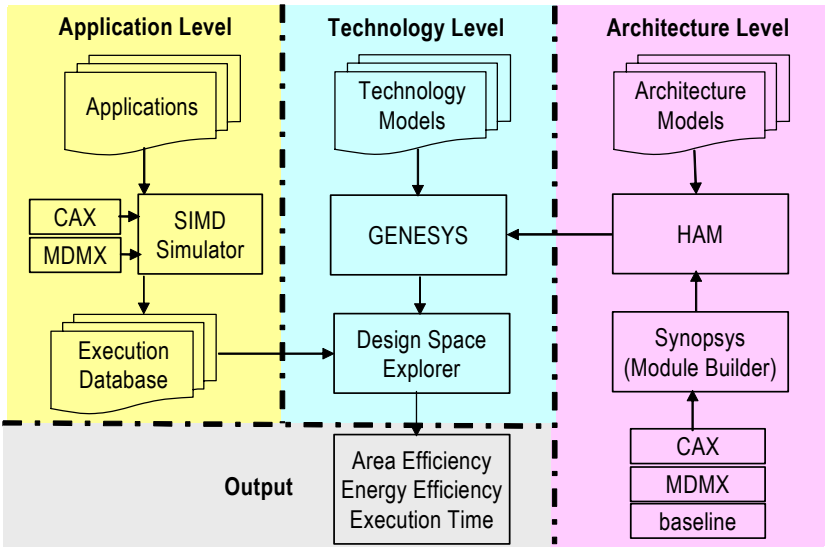


Fig. 5. A simulation infrastructure for exploring the design space of modeled architectures

7 System Area and Power Evaluation Using Technology Modeling

This section identifies the system area and power of each SIMD configuration using technology modeling. The GENESYS tool [19] has been used to determine implementation characteristics (e.g., system area and power) for each PE configuration. Figures 6 and 7 show system area and power estimations versus

VPPEs, respectively, in which all the configurations were examined in the same 100nm technology and 50 MHz node frequency. For VPPEs above or at 64, both system area and power asymptotically approach a lower limit where local memory area dominates. Below this point, however, the system area and power decrease exponentially. As a result, a number of configurations are not feasible, requiring silicon area greater than 1,000 mm^2 (the ITRS [12] projected limit in 100 nm CMOS technology). Although some configurations with power above three watts are not feasible as well in terms ofbattery operation and heat removal, power reduction techniques [1] (e.g., clock frequency scaling) allow the power dissipation levels required by portable, battery-operated devices at the expense of performance (execution time). These system area and power results are combined with application simulations to determine both area and energy efficiency for each case, which is presented next.

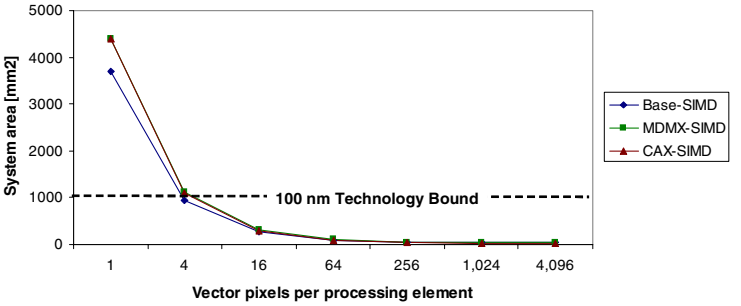


Fig. 6. System area versus VPPE

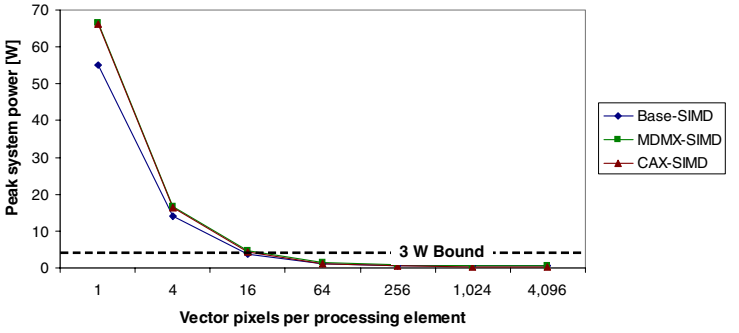


Fig. 7. Peak system power versus VPPE

8 Experimental Results

Application simulation and technology modeling are used to determine performance and efficiency for each SIMD configuration for FSVQ. The execution time, sustained throughput, area efficiency, and energy consumption of each case form the basis of the study comparison, defined in Table 2.

Table 2. Summary of evaluation metrics

execution time	sustained throughput	area efficiency	energy consumption
$t_{exec} = \frac{C}{f_{ck}}$	$Th_{sust} = \frac{O_{exec} \cdot U \cdot N_{PE}}{t_{exec}}$	$\eta_A = \frac{Th_{sust}}{Area}[\frac{\text{Gops}}{\text{s} \cdot \text{mm}^2}]$	$Energy = Power \cdot t_{exec}[J]$

C is the cycle count, f_{ck} is the clock frequency, O_{exec} is the number of executed operations, U is the system utilization, and N_{PE} is the number of processing elements. Note that since each CAX and MDMX instruction executes more operations (typically six and three times, respectively) than a baseline instruction, we assume that each CAX, MDMX, and baseline instruction executes six, three, and one operation, respectively, for the sustained throughput calculation.

8.1 Execution Performance Evaluation Results

This section evaluates the effects of different VPPE ratios on processing performance for each case. The impact of CAX on each VPPE configuration is also presented.

Effects of Varying VPPE Ratios on Processing Performance. Figure 8 shows execution performance (in sustained throughput) for different VPPE configurations with and without CAX or MDMX. As expected, the sustained throughput increases as the VPPE ratio decreases because of more data parallelism (or an increase in available processing elements). Most VPPE configurations except for at VPPE = 4,096 are capable of delivering in excess of 1 Gops/s required by an MPEG-2 encoded video stream at video rate (30 frame/s) [8].

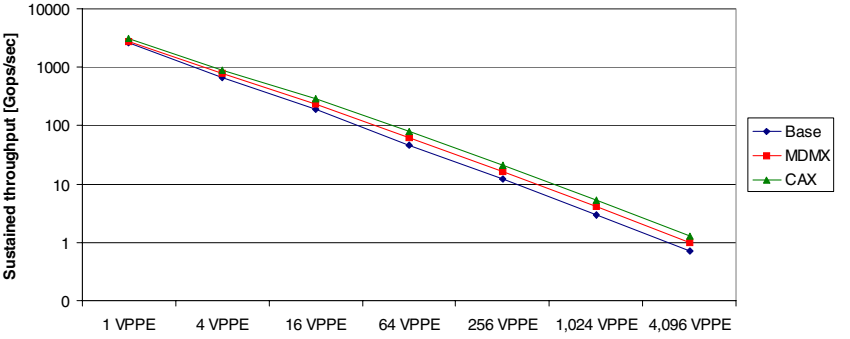


Fig. 8. Sustained throughputs for different VPPE configurations with and without CAX or MDMX

Impact of CAX on Different VPPE Configurations. Figure 9 shows the distribution of the vector instructions for each VPPE configuration with CAX and MDMX, normalized to the baseline version without subword parallelism. Each bar divides the instructions into the arithmetic-logic-unit (ALU), memory (MEM), communication (COMM), PE activity control unit (MASK), image pixel loading (PIXEL), MDMX,

and CAX. Results indicate that the instruction count decreases from 29.6% (at VPPE = 1) to 89.4% (at VPPE = 4,096) with CAX, but only 24.8% (at VPPE = 1) to 83.6% (at VPPE = 4,096) with MDMX over the baseline version. An interesting observation is that for VPPEs below 16 both CAX and MDMX are less effective at reducing vector instructions. This is because high inter-PE communication operations are involved in the task, which are not affected by CAX or MDMX.

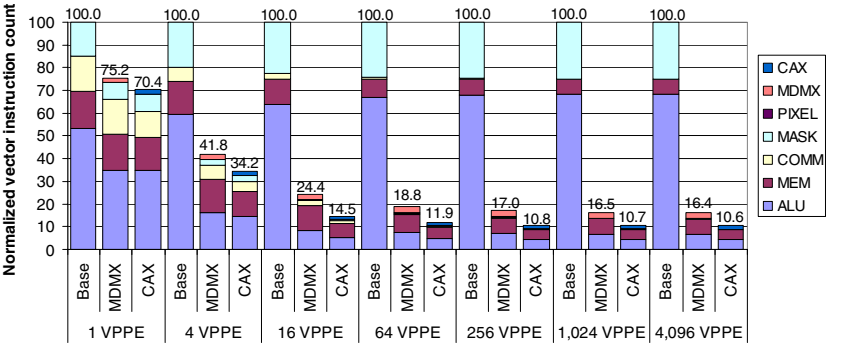


Fig. 9. Issued vector instructions for each VPPE configuration with MDMX and CAX, normalized to the baseline version

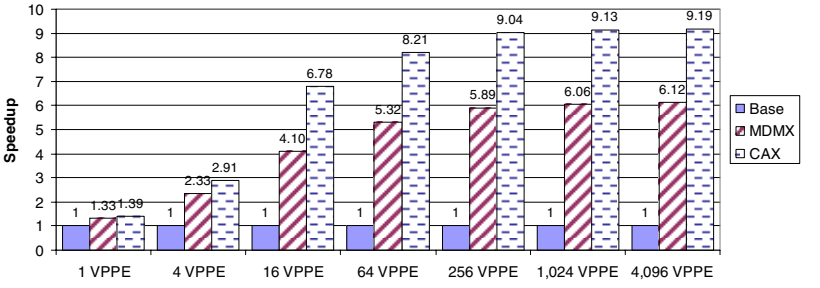


Fig. 10. Speedups of each VPPE configuration with CAX and MDMX, normalized to the baseline performance

Figure 10 presents additional data showing speedups for each VPPE configuration with CAX and MDMX, normalized to the baseline performance. CAX outperforms MDMX for all the VPPE configurations in speedup, indicating 1.4× (at VPPE = 1) to 9.2× (at VPPE = 4,096) with CAX, but only 1.3× (at VPPE = 1) to 6.1× (at VPPE = 4,096) with MDMX over the baseline version. Table 3 summarizes simulation results for each case. Since size and battery life are as critical for embedded systems as performance, area- and energy-related results are discussed next.

Table 3. Application performance of the baseline, MDMX, and CAX versions on a 3-band 256 × 256 pixel system running at 50 MHz. Note that system utilization is calculated as average concurrency, and it is relative to total system size.

# of VPPE	ISA	Vector Instruction	Scalar Instruction	System Utilization [%]	Execution Time [msec]	Sustained Throughput [Gops/sec]
1 VPPE	Baseline	34,101	14,873	80.8	0.68	2,646
	MDMX	25,653	14,873	82.1	0.51	2,798
	CAX	24,014	11,609	82.0	0.48	3,116
4 VPPE	Baseline	59,392	18,731	82.8	1.19	678
	MDMX	24,832	18,731	87.3	0.50	789
	CAX	20,302	13,343	85.1	0.41	873
16 VPPE	Baseline	183,860	20,755	92.0	3.68	188
	MDMX	44,850	20,755	96.0	0.90	235
	CAX	26,602	12,259	93.9	0.53	285
64 VPPE	Baseline	684,689	49,707	92.3	13.69	47
	MDMX	128,657	49,707	96.4	2.57	62
	CAX	81,361	28,203	94.2	1.63	79
256 VPPE	Baseline	2,674,449	164,483	92.0	53.49	12
	MDMX	454,417	164,483	96.2	9.09	16
	CAX	287,761	91,779	94.0	5.76	21
1,024 VPPE	Baseline	10,634,161	623,469	92.0	212.68	2.9
	MDMX	1,754,033	623,469	96.2	35.08	4.1
	CAX	1,132,513	347,013	94.0	22.65	5.2
4,096 VPPE	Baseline	42,464,544	2,455,523	91.9	849.29	0.7
	MDMX	6,944,032	2,455,523	96.2	138.88	1.0
	CAX	4,488,368	1,364,907	94.0	89.77	1.3

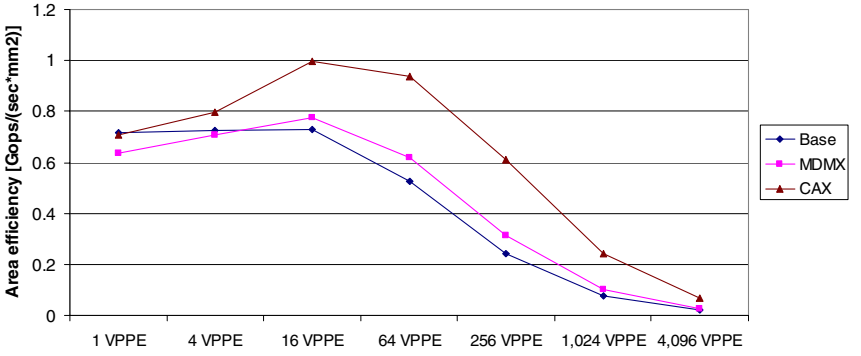


Fig. 11. Area efficiency versus VPPE

Area-Related Evaluation Results. Figure 11 presents area efficiency for each case. All three versions achieve their maximum area efficiency at VPPE = 16 due to the inherent definition of FSVQ. For VPPEs above or at 16, the area efficiency decreases

almost linearly because the number of operations to perform the task increases more rapidly with VPPE than the area level at which local memory area dominates.

Energy-Related Evaluation Results. Figure 12 presents energy consumption for each VPPE configuration with MDMX and CAX, normalized to the baseline version. Each bar divides energy consumption into the functional unit (FU, combines ALUs, Barrel Shifter, and MACC), storage (combines Register file and Memory), and others (combines Comm., Sleep, Serial, and Decoder) categories. The results indicate that energy consumption is reduced from 26% (at VPPE = 1) to 89% (at VPPE = 4,096) with CAX, but only 24% (at VPPE = 1) to 84% (at VPPE = 4,096) with MDMX over the baseline. For VPPEs below 16, both MDMX and CAX are less efficient at reducing energy consumption because of the smaller reduction rate in the instruction count.

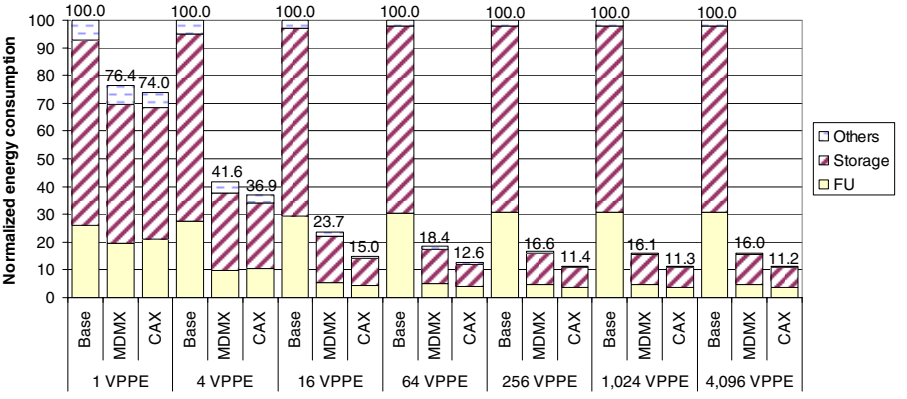


Fig. 12. Energy consumption for each VPPE configuration with CAX and MDMX, normalized to the baseline version

9 Conclusions

Continued advances in multimedia computing will rely on reconfigurable silicon area usage within an integrated pixel processing array. This paper has explored the effects of varying the VPPE ratio (number of vector pixels mapped to each processor within a SIMD architecture). Moreover, the impact of CAX on each VPPE configuration has been evaluated to identify the most efficient PE granularity for a specified SIMD array and implementation technology. Experimental results using architectural and workload simulation indicate that as the VPPE value decreases, the sustained throughput increases due to more data parallelism. However, the most efficient VPPE ratio does not occur at VPPE = 1. Results suggest that the most efficient operation for 3-D vector quantization is achieved at VPPE = 16 with CAX, delivering in excess of 280 gigaoperations per second using 280 mm^2 of silicon area and 4.2W of power for 4,096 PEs running at 50 MHz clock frequency and 100 nm technology.

References

1. A. Bellaouar and M. I. Elmasry: Low-Power Digital VLSI Design: Circuits and Systems. Boston: Kluwer Academic (1995)
2. S. Bond, S. Jung, O. Vendier, M. Brooke, N. M. Jokerst, S. Chai, A. Lopez-Lagunas, and D. S. Wills: 3D stacked Si CMOS VLSI smart pixels using through-Si optoelectronic interconnections. Proc. IEEE Lasers and Electro-Optics Soc. Summer Topical Meeting on Smart Pixels (1998) 27-28
3. S. M. Chai, T. M. Taha, D. S. Wills, and J. D. Meindl: Heterogeneous architecture models for interconnect-motivated system design. IEEE Trans. VLSI Systems, special issue on system level interconnect prediction, vol. 8, no. 6 (2000) 660-670
4. Coding of Moving Pictures and Audio. ISO/IEC JTC1/SC29/WG11 N3312 (2000)
5. E. R. Fossum: Digital camera system on a chip. IEEE Micro, vol.18, no. 3 (1998) 8-15
6. E. R. Fossum: Architectures for Focal Plane Image Processing. Optical Engineering, vol. 28, no. 8 (1989) 865-871
7. A. Gentile, S. Sander, L. M. Wills, and D. S. Wills: Impact of pixel to processing ratio in embedded SIMD image processing architectures. Journal of Parallel and Distributed Computing. vol. 64, no. 11 (2004) 1318-1332
8. A. Gersho and R. M. Gray: Vector Quantization and Signal Compression. Kluwer Academic (1992)
9. R. C. Gonzalez and R. E. Woods: Digital Image Processing. Prentice Hall (2002)
10. H.-M. Hang and B. G. Haskell: Interpolative vector quantization of color images. IEEE Transactions on Communications, vol. 36, no. 4 (1988) 465-470
11. M. C. Herbordt, A. Anand, O. Kidwai, R. Sam, and C. C. Weems: Processor/ memory/ array size tradeoffs in the design of SIMD arrays for a spatially mapped workload. Proc. IEEE Intl. Workshop on Computer Architecture for Machine Perception (1997) 12-21
12. The international technology roadmap for semiconductors. http://public.itrs.net (2003)
13. J. Kim: Architectural enhancements for color image and video processing on embedded systems. PhD dissertation, Georgia Inst. of Technology (2005)
14. J. Kim and D. S. Wills: Evaluating a 16-bit YCbCr (6:5:5) color representation for low memory, embedded video processing. Proc. of the IEEE Conf. on Consumer Electronics (2005) 181-182
15. S. C. Kwatra, C. M. Lin, and W. A. Whyte: An adaptive algorithm for motion compensated color image coding. IEEE Trans. Commun., vol. COM-35 (1987) 747-754
16. R. B. Lee: Subword parallelism with MAX-2. IEEE Micro, vol. 16, no. 4 (1996) 51-59
17. MIPS extension for digital media with 3D, Technical Report http://www.mips.com, MIPS technologies, Inc., (1997)
18. MMX^{TM} Technology Technical Overview. http://www.x86.org/intel.doc/mmxmanuals.htm
19. S. Nugent, D. S. Wills, and J. D. Meindl: A hierarchical block-based modeling methodology for SoC in GENESYS. Proc. of the 15th Ann. IEEE Intl. ASIC/SOC Conf. (2002) 239-243
20. A. Peleg and U. Weiser: MMX technology extension to the Intel architecture. IEEE Micro, vol.16, no. 4 (1996) 42-50
21. K. N. Plataniotis and A. N. Venetsanopoulos: Color Image Processing and Applications. Springer-Verlag (2000)

22. J. Suh and V. K. Prasanna: An efficient algorithm for out-of-core matrix transposition. IEEE Trans. on Computers, vol. 51, no. 4 (2002) 420-438
23. M. Tremblay, J. M. O'Connor, V. Narayanan, and L. He: VIS speeds new media processing. IEEE Micro, (1996) 10-20
24. C. C. Yang: Effects of coordinate systems on color image processing. MS Thesis, University of Arizona, Tucson (1992)

A Technique to Reduce Preemption Overhead in Real-Time Multiprocessor Task Scheduling

Kyong Jo Jung and Chanik Park

System Software Laboratory,
Pohang University of Science and Technology,
Kyungbuk, Republic of Korea
{braiden, cipark}@postech.ac.kr

Abstract. Partitioning and global scheduling are two approaches for scheduling real-time tasks in multiprocessor environments. Partitioning is the more favored approach, although it is sub-optimal. This is mainly due to the fact that popular uniprocessor real-time scheduling algorithms, such as EDF and RM, can be applied to the partitioning approach with low scheduling overhead. In recent years, much research has been done on global real-time multiprocessor scheduling algorithms based on the concept of "proportionate fairness". Proportionate fair (Pfair) scheduling [5][6] is the only known optimal algorithm for scheduling real-time tasks on multiprocessor. However, frequent preemptions caused by the small quantum length for providing optimal scheduling in the Pfair scheduling make it impractical. Deadline Fair Scheduling (DFS) [1] based on Pfair scheduling tried to reduce preemption-related overhead by means of extending quantum length and sharing a quantum among tasks. But extending quantum length causes a mis-estimation problem for eligibility of tasks and a non-work-conserving problem.

In this paper, we propose the Enhanced Deadline Fair Scheduling (E-DFS) algorithm to reduce preemption-related overhead. We show that E-DFS allows us to decrease quantum length by reducing overhead and save wasted CPU time that is caused by preemption-related overhead and miss-estimation of eligibility.

1 Introduction

Recent advances in computing and communication technologies have led to a proliferation of demanding multimedia applications such as streaming audio, video players, and multi-player games. According to rise of interest in applications requiring real-time constraints and predictable performance guarantees, much research has been done on design of multiprocessor servers and real-time multiprocessor scheduling.

Real-time multiprocessor task scheduling techniques fall into two general categories, partitioning and global scheduling. Under partitioning, each processor has own local scheduler and local ready queue. Each processor schedules tasks independently from a local ready queue and each task is assigned to a particular

T. Srikanthan et al. (Eds.): ACSAC 2005, LNCS 3740, pp. 566–579, 2005.

processor. In contrast, all ready tasks are stored in a single queue under global scheduling. The global scheduler assigns tasks to processors.

Partitioning is the more favored approach because popular uniprocessor real-time scheduling algorithms, such as earliest-deadline-first (EDF) and rate-monotonic (RM) algorithms[1], can be applied to the partitioning approach with low scheduling overhead. Partitioning, regardless of the scheduling algorithm used, has two primary flaws. First, it is suboptimal when scheduling periodic tasks. A well-known example is a two-processor system that contains three synchronous periodic tasks, each with an execution cost of 2 and a period of 3. All tasks in such a system is impossible without migration. Hence, this task set is not schedulable under the partitioning approach. Second, the assignment of tasks to processors is a NP-hard problem. Hence, optimal task assignments cannot be obtained online due to the run-time overhead. Online partitioning is typically done using heuristics, which may be unable to schedule task systems that are schedulable using offline partitioning algorithms.

In recent years, much research has been done on global real-time multiprocessor scheduling algorithms that ensure fairness[1][2][4][5][9][12]. Proportionate fair (Pfair) scheduling, proposed by Baruah *et al.*[6], is the only known optimal algorithm for scheduling real-time tasks on multiprocessors. However, frequent preemptions caused by the small quantum length for providing optimal scheduling in Pfair scheduling make the Pfair scheduling impractical. Deadline Fair Scheduling (DFS) [1] based on the Pfair scheduling tried to reduce preemption-related overhead by means of extending quantum length and sharing a quantum among tasks. But extending quantum length causes a mis-estimation problem for eligibility of tasks and a non-work-conserving problem.

In this paper, we propose the Enhanced Deadline Fair Scheduling (E-DFS) algorithm to reduce preemption-related overhead. We show that E-DFS allows us to decrease quantum length by reducing overhead and save wasted CPU time that is caused by preemption-related overhead and miss-estimation of eligibility.

The rest of this paper is structured as follows. Section 2 presents Pfair and DFS algorithms. Section 3 introduces the enhanced deadline fair scheduling. Section 4 presents the results of our experimental results. Finally, section 5 presents conclusions, some limitation of our approach, and directions for future work.

2 Background

2.1 Pfair Scheduling

Under Pfair scheduling, a periodic task T with an integer period $T.p$ and an integer execution cost $T.e$ has a weight $wt(T) = T.e/T.p$, where $0 < wt(T) \leq 1$. Processor time is allocated in discrete time units, called *quantum*, and each processor can be allocated to at most one task in each quantum. A task may

[1] Dhall and Liu have show that global scheduling using EDF or RM can result in arbitrarily-low processor utilization in multiprocessor systems [7].

be allocated time on different processors, but not within the same quantum. The task set τ is schedulable iff $\sum_{T \in \tau} wt(T) \leq M$. This equation is proved by Brauah *et al.*[6].

Subtask In Pfair scheduling algorithm, each task T is divided into an infinite sequence of quantum-length *subtasks*. The i^{th} subtask ($i \geq 1$) of task T is denoted T_i. Each subtask has pseudo-release time and pseudo-deadline. The pseudo-release time and the pseudo-deadline is decided by the number of received quantum (subtask's index) and the weight of task.

$$r(T_i) = \left\lfloor \frac{i-1}{wt(T)} \right\rfloor \qquad d(T_i) = \left\lceil \frac{i}{wt(T)} \right\rceil$$

By means of concepts of subtask, quantum, and release time of subtask, CPU allocation time of each task is aligned by weight. In ineligible point, the task relinquishes CPU to other task.

Challenges in Pfair Scheduling. Pfair scheduling algorithm provides optimal scheduling in a real-time multiprocessor environment. However, for optimality, Pfair scheduling requires execution cost of tasks to be a multiple of quantum length. This can lead to a large loss in schedulability. For example, assuming a quantum length of 1, if a task has a small execution cost of ϵ, then it must be increased to 1. If the task's period is 1, this would mean a schedulability loss of $1 - \epsilon$. This is not acceptable in practice.

One way to reduce schedulability loss is decreasing quantum length. However, this leads to wasting CPU time by preemption-related overhead caused by frequent preemptions. This trade-off between preemption overhead and quantum length remains an open issue in Pfair scheduling algorithm.

2.2 Deadline Fair Scheduling

Deadline Fair Scheduling(DFS) modifies Pfair scheduling to reduce preemption overhead and apply it in real system. Under DFS, each task T specifies a share ϕ_T that indicates the proportion of the processor bandwidth required by that task. Since a task can run on only one processor at a time, each task cannot ask for more than $1/M$ of the total system bandwidth, where M is the number of CPUs. Consequently, a necessary condition for feasibility of the current task set is $\frac{\phi_T}{\sum_j \phi_j} \leq \frac{1}{M}$.

Like subtask in Pfair scheduling, each task consists of a series of runs of one quantum each. DFS uses an eligibility criterion and internally generated deadline to allow each task to receive processor bandwidth based on the requested share, while ensuring that no task gets more or less that its due share in each period. The eligible condition generated by eligibility criterion is decided by received CPU time and weight like pseudo release time and pseudo deadline in Pfair. Each eligible task is stamped with an internally generated deadline. DFS schedules eligible tasks in earliest deadline first order to ensure that each task receives its due share before the end of its period.

Eligibility Criterion and Deadline. Let $m_T(t)$ be the number of times that task T has been up to time t. Let us also assume that the quantum length=1, and each task always runs for an entire quantum. With these assumptions to maintain P-fairness, we require that for all times ty and task T,

$$\left\lfloor \frac{tM\phi_T}{\sum_j \phi_j} \right\rfloor \leq m_T(t) \leq \left\lceil \frac{tM\phi_T}{\sum_j \phi_j} \right\rceil,$$

where $t \cdot M$ is the total CPU capacity on the M processors in time $[0, t)$. The eligibility requirements ensure that $m_T(t)$ never exceeds this range, and the deadlines ensure that $m_T(t)$ never falls short of this range. Thus, eligibility criterion and deadline are specified as following.

$$EligibilityCriterion : m_T(t) + 1 \leq \left\lceil (t+1)M\frac{\phi_T}{\sum_j \phi_j} \right\rceil$$

$$Deadline : \left\lceil (m_T(t)+1)\frac{\sum_{j=1}^{N} \phi_j}{\phi_T} \right\rceil$$

As mentioned above, the Pfair algorithm requires that execution cost of tasks is a multiple of quantum length and each task always runs for an entire quantum providing optimal schedule. These requirements cause preemption overhead due to the small quantum length. To remove these impractical requirements and reduce overhead, DFS extends quantum length and shares a quantum among tasks. To share a quantum, the DFS algorithm uses a method such as start tag S_T, finish tag F_T, and virtual time v that are used in WFQ[8] and SFQ[10] algorithm, for accounting for the amount of CPU service that each task has achieved.

DFS tried to apply Pfair scheduling to real systems by means of preemption overhead reduction achieved by extending quantum length and sharing it among tasks. However, extending quantum length breaks the assumption of Pfair for optimality and causes a non-work-conserving by miss-estimation problem for eligibility condition. DFS uses a heuristic approach to eliminate wasted CPU time ; it uses an auxiliary scheduler that serves runnable and ineligible tasks if the eligible queue is empty.

3 Proposed Algorithm

Although the Pfair scheduling algorithm is the only known optimal method for multiprocessor real-time scheduling, preemption-related overhead caused by frequent preemption for optimal schedule make it impractical. In this paper, we propose the Enhanced Deadline Fair Scheduling (E-DFS) algorithm to reduce preemption-related overhead including scheduling and cache-related overhead.

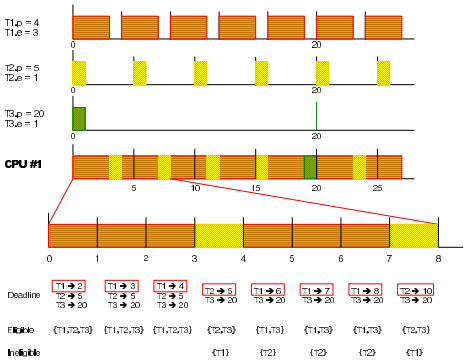


Fig. 1. Example of DFS scheduling

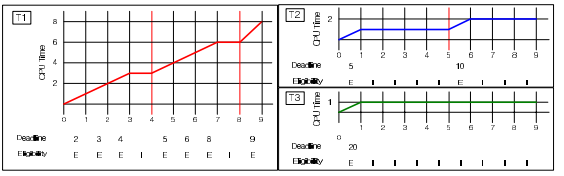


Fig. 2. Comparison with eligible point of tasks

3.1 Basic Concept of E-DFS

The basic concepts of the Pfair and DFS algorithms are that the execution point is aligned by the task's weight. That means patterns of eligible and ineligible tasks are decided by the weight of tasks.

For example, let us assume that three tasks are running on a CPU and the quantum length is 1. A weight is assigned to each task ($\phi_{T1} = 3/4, \phi_{T2} = 1/5, \phi_{T3} = 1/20$). As shown in Figure 2, the eligible and ineligible state of each task are decided by the relative weight of each task. Under DFS, the scheduler computes the set of eligible tasks according to the eligible state of tasks and picks M tasks with the earliest deadline every quantum as shown in Figure 1. Therefore the number of scheduling points is 8 in the example of Figure 1.

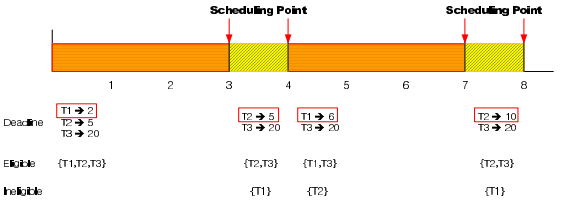


Fig. 3. Scheduling point in E-DFS scheduling

Table 1. Variables used in algorithm

variable	description
q	quantum length
M	number of processors
ϕ_i	weight of task i
ϕ_{sum}	weight sum of total tasks
S_i	start tag of task t
v	virtual time ($v = \left(\sum_j \phi_j \cdot S_j\right) / \sum_j \phi_j$)
$t_{in,i}$	Time up to ineligible point of task i
t_{SP}	Time up to scheduling point
MAX_{SP}	Maximum time up to scheduling point

The basic concept of our proposed algorithm is to preempt tasks only at the context-switch point to eliminate unnecessary scheduling points by means of using eligible patterns of each task. We can be aware of the pattern of selected M tasks by its relative weight, and compute the ineligible point. By means of these algorithms, we can preempt tasks only when we have to preempt a task due to its ineligible state. As shown in Figure 3, at time 0 task 1 is selected to schedule. Since the ineligible point of task 1 is time 3, task 1 can be scheduled up to time 3. At time 3, task 1 has to be preempted because its eligible state is ineligible. So the next scheduling point is set to time 3 and task 1 is switched to the next selected task. In Figure 3, we can reduce the number of preemptions to 4.

The algorithm to compute the next scheduling point is as Algorithm 1 and Table 1 represents the notations used in the algorithm. In the algorithm, the scheduler computes the point switched to ineligible state of selected task by means of the *eligibility criterion*. After that, the scheduler sets the next scheduling point to the computed point. At the next scheduling point, the next scheduled task is selected and it repeats the operations described before.

3.2 Comparison with Preemption-Related Overhead

The E-DFS algorithm is proposed to reduce preemption-related overhead by means of eliminating unnecessary scheduling points. In this subsection, we compare overhead cost between DFS and proposed E-DFS.

Preemption-related overhead includes scheduling overhead, context-switch overhead, and cache-related overhead [3]. *Scheduling overhead* account for the time spent moving a newly-arrived or preempted task to the ready queue and choosing the next task to be scheduled. *Context-switching overhead* accounts for the time the operating system spends on saving the context of a preempted task

Algorithm 1: Algorithm for deciding scheduling point

```
begin
    for i ← 1 to M do
        while task i is eligible and t_SP < MAX_SP do
            S_i ← S_i + q/φ_i
            v ← v + q/φ_sum
            t_in,i ← t_in,i + q

        end
    end
    t_SP ← min(t_in,1, t_in,2, ..., t_in,M)

end
```

and loading the context of the task that preempts it. *Cache-related overhead* of a task refers to the time required to service cache misses that a task suffers when it resumes after a preemption. These overhead costs waste CPU time thus delaying the execution of tasks.

Proposed E-DFS can reduce the scheduling overhead by means of eliminating unnecessary scheduling points, and reduce the context-switching and cache-related overhead by means of reducing the number of context-switches. In the case of DFS, scheduling overhead occurs in every quantum. So the scheduling cost up to time t is $\left\lceil \frac{t}{q} \right\rceil \cdot S_T$. But under E-DFS, scheduling operations are conducted only when context-switches have to be performed or a new task arrives[2]. Therefore scheduling cost up to time t under E-DFS is $(N_C(t) + N_A(t)) \cdot S_T$. After performing scheduling operations, scheduler computes the next scheduling point for selected tasks. These operations are performed at the point that context-switches and new arrival are not occurred. The number of these points is $\left\lceil \frac{t}{q} \right\rceil - (N_C(t) + N_A(t))$. Consequently the cost of computing eligible condition of seletected tasks up to next scheduling point is $\left\lceil \frac{t}{q} \right\rceil - (N_C(t) + N_A(t)) \cdot S_O$.

Using notations of Table 2, total preemption-related costs up to time t in the cases of DFS and E-DFS are computed as Equation 1 and Equation 2 each.

$$\left\lceil \frac{t}{q} \right\rceil \cdot S_T + N_C(t) \cdot (C + D) \tag{1}$$

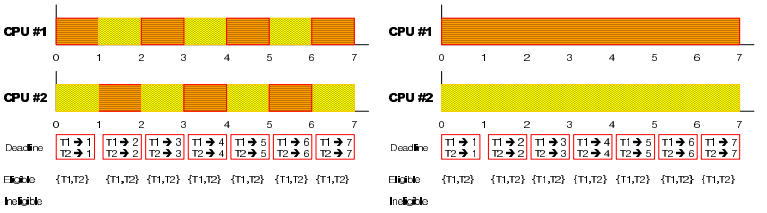
$$(N_C(t) + N_A(t)) \cdot S_T + \left(\left\lceil \frac{t}{q} \right\rceil - (N_C(t) + N_A(t)) \right) \cdot S_O + N_C(t) \cdot (C + D) \tag{2}$$

[2] If new task is arrived, we have to re-compute next scheduling point because total weight sum is changed.

[3] In the case that context-switch occurred by arrival task, the number of arrival tasks is included in $N_C(t)$.

Table 2. Notations for Comparison of Overhead

Variable	Description
S_T	Total scheduling cost for whole runnable tasks
S_O	Scheduling cost per task
D	Cache-related overhead
C	Context-switching overhead
$N_C(t)$	Number of context-switches up to time t
$N_A(t)$	Number of arrival tasks up to time t without context-switch [3]



(a) Context-switches in DFS (b) Context-switches in E-DFS

Fig. 4. Comparison with context-switches in worst case

In worst case, if context-switches are occurred in every quantum ($\left\lceil \frac{t}{q} \right\rceil = N_C(t) + N_A(t)$), total costs of DFS and E-DFS are equivalent. In average case, if ($\left\lceil \frac{t}{q} \right\rceil \geq N_C(t) + N_A(t)$), we can save scheduling cost of $\left(\left\lceil \frac{t}{q} \right\rceil - (N_C(t) + N_A(t)) \right) \cdot (S_T - S_O)$. Another advantage under E-DFS is saving the context-switching and the cache-related overhead by reducing the number of context-switches. Consider synchronous tasks with same weight. In the worst case, if the DFS chooses a task randomly for tasks with the same deadline, the DFS can suffer from context-switches in every quantum because the same weight induces the same deadline. Under E-DFS, since a selected task can be executed up to the next scheduling point without a context-switch, the number of context-switches can be reduced compared with the case of DFS. For example, let us assume that two tasks with the same weight are running on two CPUs. As shown in Figure 4, DFS can suffer context-switches every quantum in the worst case, while E-DFS can schedule tasks without unnecessary context-switches.

4 Experimental Result

4.1 Implementation of E-DFS

We implemented the E-DFS algorithm into the Linux kernel 2.6. The E-DFS scheduler replaces the standard time-sharing scheduler in Linux. Our implemen-

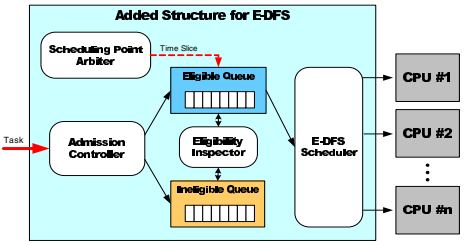


Fig. 5. E-DFS Scheduler

tation allows each task to specify a weight ϕ_i. Tasks can dynamically change or query their weights using two new system calls, *setweight* and *getweight*. Their interface is similar to the Linux system calls *setpriority* and *getpriority*. The components of the E-DFS algorithm consist of the following.

- `Admission Controller` : Check feasibility of task set
- `Eligibility Inspector` : Determine if eligibility of any tasks has become changed
- `Scheduling Point Arbiter` : Compute the next scheduling point and allocate the time slice which is the amount of time until next scheduling point

We added two run queues - one for eligible tasks and the other for ineligible tasks as shown in Figure 5. The eligible queue consists of tasks sorted in deadline order. The E-DFS scheduler services these tasks using EDF. Once eligible, a task is removed from the ineligible queue and inserted into the eligible queue. Whenever an allocated time slice of a task blocks for I/O or departs, the kernel invokes the E-DFS scheduler. The scheduler first updates the start tag and finish tag of the task. Next, it recomputes the virtual time based on the start tags of all the runnable tasks. It determines if any ineligible tasks have become eligible, and if so, moves them from the ineligible queue to the eligible queue in deadline order. The scheduler then picks M tasks at the head of the eligible queue and schedules it for execution. Finally, the *Scheduling Point Arbiter* computes the next scheduling point based on eligible patterns of selected tasks, and assign time slice, which is the amount of time up to the next scheduling point, to selected tasks.

4.2 Experimental Setup

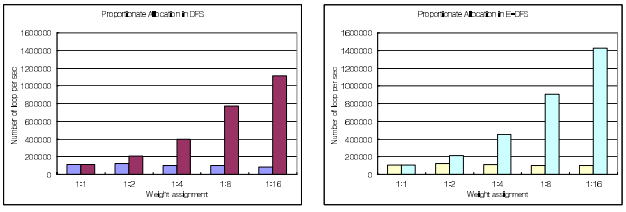
We conducted experiments to (i) demonstrate the proportionate allocation property of E-DFS, (ii) measure the preemption-related overhead, and (iii) analyze experimental results.

For our experiments, we used a 2.4 GHz Intel XEON based dual-processor with Hyper-Threading and 512 KB cache. Main memory is 1GB and quantum length is 1ms. The workload for our experiments consisted of a combination of benchmarks and sample applications that we wrote to demonstrate specific feature. These applications include: (i) *dhdystone*, a compute-intensive benchmark

for measuring integer performance, (ii) *lmbench* [11], a benchmark that measures various aspects of operating system performance, and (iii) *Inf*, a compute-intensive application that performs computations in an infinite loop.

4.3 Proportionate Allocation

We first demonstrate that the E-DFS allocates processor bandwidth to applications in proportion to their weight. We conducted an experiment with a number of *dhrystone* tasks. We ran two dhrystone applications with relative weights of 1:1, 1:2, 1:4, 1:8,and 1:16 in the presence of 30 background dhrystone applications using both the DFS and E-DFS algorithms. In Figure 6, x-axis represents assigned weight and y-axis represents the number of loops per sec. The two applications receive processor bandwidth in proportion to the specified weight in both algorithms. However, in the case of E-DFS, the number of loops is increased compared with the case of DFS. This result was achieved through saving CPU time by means of reducing preemption-related overhead. Figure 7 shows the number of context-switches in the first experiment. In the case of relative weight of 1:16, the number of context-switches is decreased to 57% in E-DFS. In that case, the task with weight of 16 have to use about 25% processor bandwidth because the average weight sum is 70. Since we used dual-processors with Hyper-Threading, that task had to almost always be executed on one CPU to satisfy processor allocation in proportion to its weight. Under DFS, since other tasks that except the task with weight 16 have the same weight of 1 and the scheduling decision is performed every quantum, the task with weight of 16 and tasks with weight of 1 are executed by turns. The E-DFS scheduler can cause less



(a) DFS Scheduler (b) E-DFS Scheduler

Fig. 6. Proportionate Allocation

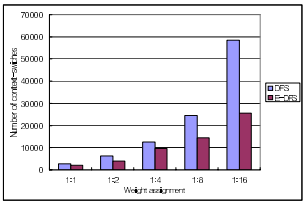
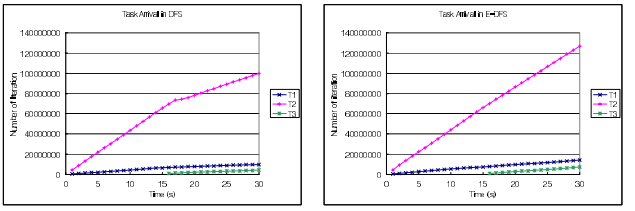


Fig. 7. Comparison with the number of context-switches



(a) DFS scheduler (b) E-DFS scheduler

Fig. 8. Proportionate allocation using *Inf* application in dynamic task set

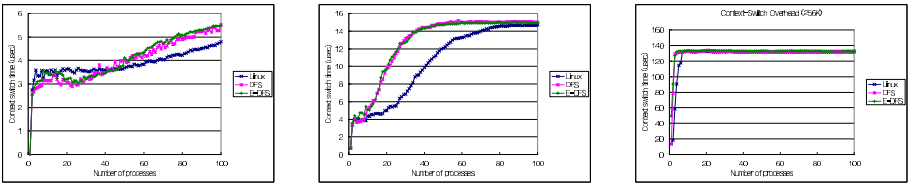
context-switches compared with DFS because the selected task can be executed to the next scheduling point in E-DFS. Therefore CPU time saved by reducing preemption overhead can be assigned to user tasks.

4.4 Proportionate Allocation in Dynamic Task Set

We performed a second experiment to show proportionate allocation in a dynamic task set. At t=0, we started two *Inf* applications ($T1$ and $T2$) with weight 1:10. At t=15s, we started a third *Inf* application ($T3$) with a weight of 1. Figure 8 shows the results of second experiment. As expected, each scheduler allocates CPU time to processes in proportion to their weights. As with the results of the first experiment, E-DFS assigns the CPU time to processes by reducing preemption overhead.

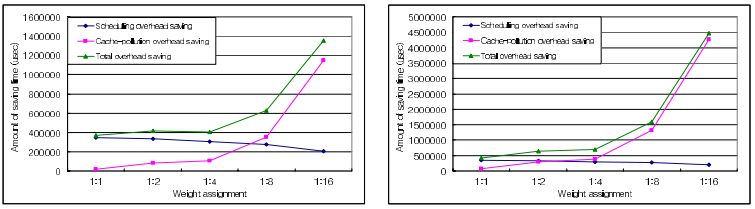
4.5 Preemption-Related Overhead

We measured preemption-related overhead imposed by the E-DFS scheduler using *lmbench* benchmark. We ran lmbench on a lightly loaded machine with E-DFS and repeated the experiment with the Linux time sharing scheduler and DFS scheduler. In each case, we averaged the statistics reported by lmbench over several runs to reduce experimental error.



(a) Context-switching overhead imposed by 0KB processes (b) Context-switching overhead imposed by 16KB processes (c) Context-switching overhead imposed by 256KB processes

Fig. 9. Context-switching overhead varying memory size per process in Linux, DFS, and E-DFS



(a) 4KB processes (b) 256KB processes

Fig. 10. Comparison of saving time by reduction of scheduling and cache overhead

Figure 9 plots the context switch overhead imposed by the three schedulers for a varying number processes. We set the array sizes manipulated by each processes to 0, 16,and 256 KB to measure scheduling overhead and cache overhead. In the case of 0 KB processes, since memory size manipulated by each processes is 0 KB, cache overhead is eliminated. Therefore Figure 9(a) plots the scheduling overheads per scheduling point. The scheduling overheads of DFS and E-DFS are similar to each other, while they are a little high compared with scheduling overhead of the Linux time-sharing scheduler. As memory sizes manipulated by each processes are increased, the difference among context-switching overheads of each schedulers become more decreased because most of context-switching overhead is occupied by the cache overhead.

4.6 Analysis of Experimental Result

In this subsection, we analyze experimental results and compare them with expected preemption overhead. As mentioned in Section 3, preemption cost in DFS and E-DFS is expected to Equation 1 and Equation 2 each. Figure 10 shows expected saving time by reducing preemption overhead using parameters measured in the experiments. We obtained data by means of substituting parameters, such as scheduling cost, cache cost, and the number of context switches, to the equations. As can be shown in Figure 10, the more memory allocated to a process, the more cache overhead is increased. So, in Figure 10(b),

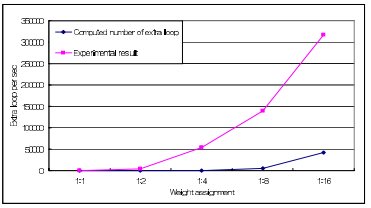


Fig. 11. Comparison between expected the number of extra loops and experimental result

most of the preemption overhead is occupied by cache overhead. By means of these data, we compared the expected number of extra loops with the result in the first experiment. Figure 11 shows comparison between expected data and experimental result. It shows that experimental result repeats more extra loops over approximately 6 times compared with the expected extra loops. This difference is caused by the difference in cache overhead. The *dhrysotne* application used in our experiment is memory-intensive application. It used approximately 4.5 MB of memory in the experiment, while Figure 11 shows the case of memory sizes manipulated by processes of 256 KB. Therefore we estimated that the smaller cache overhead applied to our expected data causes the error.

5 Conclusion and Future Work

Although the Pfair scheduling algorithm is the only known optimal method for multiprocessor real-time scheduling, it requires assumptions that "execution cost of all tasks is a multiple of quantum length" and "each task always runs for an entire quantum". The small quantum length needed to satisfy these requirements causes preemption overhead by frequent preemptions.

The DFS algorithm modifies the Pfair to reduce preemption overhead by means of extending and sharing a quantum. However, extending quantum length causes a non-work-conserving problem by mis-estimation for eligibility. The DFS tried to solve this problem heuristically.

In this paper, E-DFS is proposed to reduce preemption-related overhead in real-time multiprocessor scheduling. The proposed algorithm reduces both scheduling overhead and cache overhead. Experimental results revealed that preemption-related overhead can be reduced effectively and increase tasks performance by means of the proposed algorithm.

Although E-DFS provides an enhanced algorithm for real-time multiprocessor scheduling, the quantum length is limited by the timer-interrupt interval. For eliminating limitation, we are interested in investigating a more sophisticated algorithm.

Acknowledgements

The authors would like to thank the Ministry of Education of Korea for its support towards the Electrical and Computer Engineering Division at POSTECH through the BK21 program. This research has also been supported in part by HY-SDR IT Research Center, in part by the grant number R01-2003-000-10739-0 from the basic research program of the Korea Science and Engineering Foundation, in part by the regional technology innovation program of the Korea Institute of Industrial Technology Evaluation and Planning, and in part by the next generation PC program of the Korea ETRI.

References

1. A. Chandra, M. Adler, and P. Shenoy, "Deadline fair scheduling: Bridging the theory and practice of proportionateair scheduling in multiprocessor servers," in *Proceedings of the 7th IEEE Real-time Technology and Applications Symposium*, pages 3-14, May 2001.
2. A. Chandra, M. Adler, P. Goyal, and P. Shenoy. "Surplus fair scheduling: A proportional-share CPU scheduling algorithm for symmetric multiprocessors," in *Proceedings of the 4th ACM Symposium on Operating System Design and Implementation*, pages 45-58, October 2000.
3. A. Srinivasan, P. Holman, J. Anderson, and S. Baruah. "The case for fair multiprocessor scheduling," in *Proceedings of the 11th International Workshop on Parallel and Distributed Real-Time Systems*, April 2003.
4. J. Anderson and A. Srinivasan, "Early-release fair scheduling," in *Proceedings of the 12th Euromicro Conference on Real-time Systems*, pages 35-43, June 2000.
5. J. Anderson and A. Srinvasan, "Pair scheduling: Beyond periodic task systems," in *Proceedings of 7th International Conference on Real-time Computing Sysmtems and Applications*, pages 297-306, December 2000.
6. S. Baruah, N. Cohen, C. G. Plaxton, and D. Varvel, "Proportioinate progress: A notion of fairness in resource allocation," *Algorithmica*, 15:600-625, 1996.
7. S. Dhall and C. Liu, "On a real-time scheduling problem," *Operations Research,* 26(1):127-140, 1978.
8. A. Parekh and R. Gallagher, "A generalized processor sharing approach to flow-control in integrated services networks: The single-node case," *IEEE/ACM Transactioins on Networking*, 1(3):344-357, 1993.
9. J. Anderson and A. Sriniasan, "Mixed Pair/ERfair scheduling of asynchronous periodic taks," in *Proceedings of the 13th Euromicro Conference on Real-time Systems*, pages 76-85, 2001.
10. P. Goyal, X. Guo, and H.M. Vin, "A hierarchical CPU scheduler for multimedia operating systems," in *Proceedngs of Operating System Design and Implementation*, pages 107-122, October 1996.
11. L. McVoy and C. Staelin, "Lmbench: Portable tools for performance analysis," in *Proceedings of USENIX'96 Technical Conference, Available from http://www.bitmover.com/lmbench*, January 1996.
12. P. Holman and J.H. Anderson, "Implementing Pfairness on a symmetric multiprocessor," in *Proceedings of the 10th IEEE Real-Time and Embedded Technology and Applications Symposium*, May 2004.

Minimizing Power in Hardware/Software Partitioning

Jigang Wu, Thambipillai Srikanthan, and Chengbin Yan

Centre for High Performance Embedded Systems,
School of Computer Engineering,
Nanyang Technological University, Singapore, 639798
{asjgwu, astsrikan}@ntu.edu.sg

Abstract. Power efficiency is one of the major considerations in the current hardware/software co-designs. This paper models hardware/software partitioning as an optimization problem with the objective of minimizing power consumption. An efficient heuristic algorithm running in $O(n \log n)$ is proposed by extending the idea of solving the 0-1 knapsack problem. Also, an exact algorithm based on dynamic programming is proposed to produce the optimal solution in $O(n \cdot \mathcal{A} \cdot \mathcal{E})$ for n code fragments under the constraints: hardware area $\mathcal{A}$ and execution time $\mathcal{E}$. Computational results show that the approximate solution produced by the proposed heuristic algorithm is nearly optimal in comparison to the optimal solution produced by the proposed exact algorithm.

Keywords: hardware/software partitioning, dynamic programming, algorithm, complexity.

1 Introduction

Most modern electronic systems are composed of both hardware and software. When designing such mixed system of hardware and software (HW/SW), we will face one of the key problems called HW/SW partitioning, defined as finding a design implementation to minimize the cost of all the specification requirements. It has been shown that the efficient techniques for partitioning can achieve results in performance or power superior to software-only solution because software is more flexible and cheaper, but hardware is less power and faster.

The general partitioning problem is known to be NP-complete. Significant research work has been done in recent years. The traditional approaches include hardware-oriented and software-oriented. Hardware-oriented approach starts with a complete hardware solution and iteratively moves parts of the system to the software as long as the performance constraints are fulfilled [1], while software-oriented approach starts with a software program moving pieces to hardware to improve speed until the time constraint is satisfied [2]. In addition, many approaches emphasis the algorithmic aspects. For example, genetic algorithms are used in [3, 4] to perform system partitioning that includes hardware space exploration; Integer programming approaches are employed in [5, 6]

T. Srikanthan et al. (Eds.): ACSAC 2005, LNCS 3740, pp. 580–588, 2005.

to perform a hardware extraction approach; Simulated annealing algorithms are applied in [7]. An algorithm for 0-1 knapsack problem is transferred to solve the HW/SW partitioning with independent tasks [8]. A dynamic programming algorithm, denoted as PACE, is employed in the LYCOS co-synthesis system [9] for path-based HW/SW partitioning. All these approaches focus on minimizing the execution time of the system. Although they can work perfectly within their own co-design environments, it is not possible to compare the results obtained, because of the large differences in the co-design environments and the lack of benchmarks [10].

Power efficiency is one of the major considerations in embedded co-designs. In this paper, HW/SW partitioning is modelled as an optimization problem to minimize power consumption under two given constraints – hardware area and execution time. A heuristic algorithm, referred to as HEU, is proposed by extending an efficient idea for 0-1 knapsack problem. The performance of the heuristic algorithm is evaluated by an exact algorithm, denoted as DPP, also proposed in this paper. The computational results show that the approximate solutions are nearly optimal.

2 Problem Formulation

As in [9], the given application in this paper is also assumed to be a sequence of n basic scheduling blocks (see Fig. 1), denoted as $P = \langle B_1, B_2, \cdots, B_n \rangle$. Each block may be moved between hardware and software. B_i is followed by B_{i+1} for $i = 1, 2, \cdots, n-1$. The following notations are used to formulate the partitioning problem.

- a_i denotes the area penalty of moving B_i to hardware.
- p_i^s denotes the power required by B_i in software implementation.
- p_i^h denotes the power required by B_i in hardware implementation.
- e_i^s denotes the execution time of B_i in software implementation.
- e_i^h denotes the execution time of B_i in hardware implementation.

All these parameters can be generated by employing one of the estimation tools, *e.g.*, LYCOS in [9]. Fig. 1 shows an example of the system model with 4 blocks, where p_i represents the power saving of moving B_i to hardware, *i.e.*, $p_i = p_i^s - p_i^h$, $1 \leq i \leq 4$.

Let H denote the set of blocks assigned to hardware and S denote the set of blocks assigned to software. Our objective is finding the partitioning of P such that $P = H \cup S$ and $H \cap S = \emptyset$, which yields the minimal power consumption while having a total area penalty less than or equal to the available hardware controller area $\mathcal{A}$ and having an execution time less than or equal to the given constraint $\mathcal{E}$. In this paper $\mathcal{A}$ is called *area constraint* and $\mathcal{E}$ is called *time constraint*.

Formally, let $(x_1, x_2, \cdots, x_n)$ be a feasible solution of the partitioning problem, where $x_i \in \{1, 0\}$, $x_i = 1$ ($x_i = 0$) indicates that B_i is assigned to hardware

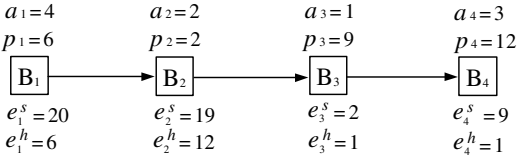


Fig. 1. The system model used by the partitioning algorithms

(software), $1 \leq i \leq n$. Thus, that the area penalty of the feasible solution is limited by $\mathcal{A}$ corresponds to

$$\sum_{i=1}^{n} a_i x_i \leq \mathcal{A},$$

and that the execution time of the feasible solution is limited by $\mathcal{E}$ corresponds to

$$\sum_{i=1}^{n} [e_i^s - (e_i^s - e_i^h)x_i] \leq \mathcal{E}, \quad \text{i.e.,} \quad \sum_{i=1}^{n} e_i x_i \geq \mathcal{E}', \text{ where } e_i = e_i^s - e_i^h \text{ and}$$

$\mathcal{E}' = \sum_{i=1}^{n} e_i^s - \mathcal{E}$. In addition, the power required by the feasible solution is

$$\sum_{i=1}^{n} [p_i^s - (p_i^s - p_i^h)x_i], \quad i.e., \quad \sum_{i=1}^{n} p_i^s - \sum_{i=1}^{n} (p_i^s - p_i^h)x_i.$$

It is clear that

$$\min \sum_{i=1}^{n} [p_i^s - (p_i^s - p_i^h)x_i] \quad \Longleftrightarrow \quad \max \sum_{i=1}^{n} (p_i^s - p_i^h)x_i.$$

Noting that $p_i = p_i^s - p_i^h$, $i = 1, 2, \cdots, n$, the problem discussed in this paper can be formulated as the following maximization problem $\mathcal{P}$ for the given $\mathcal{A}$ and $\mathcal{E}$:

$$\mathcal{P} \begin{cases} \text{maximize} & \sum_{i=1}^{n} p_i x_i, \\ \text{subject to} & \sum_{i=1}^{n} a_i x_i \leq \mathcal{A}, \\ & \sum_{i=1}^{n} e_i x_i \geq \mathcal{E}', \quad \text{and} \quad x_i \in \{0, 1\}. \end{cases}$$

This problem is one kind of the 0-1 programming problems of NP-hard. To our best knowledge, no previous algorithms can be directly used due to the mixed constraints ($\leq$ and $\geq$). Noting that the objective function and one of the constraints (the area constraint) form the standard 0-1 knapsack problem [11], we develop a heuristic algorithm and an exact algorithm for solving $\mathcal{P}$ based on the techniques of solving the 0-1 knapsack problems.

3 The Proposed Algorithms

3.1 Heuristic Algorithm

First of all, we review the knapsack problem because the HW/SW partitioning problem in this paper can be considered as an extension of 0-1 knapsack problem. Given a knapsack capacity C and the set of items $S = \{1, 2, \cdots, n\}$, where each item has a weight w_i and a benefit b_i. The problem is to find a subset $S' \subset S$, that maximizes the total profit $\sum_{i \in S'} b_i$ under the constraint that $\sum_{i \in S'} w_i \leq C$, i.e., all the items fit in a knapsack of carrying capacity C. This problem is called the knapsack problem. The 0-1 knapsack problem is a special case of the general knapsack problem defined above, where each item can either be selected or not selected, but cannot be selected fractionally. Mathematically, it can be described as follows,

$$\begin{cases} \text{maximize } \sum_{i=1}^{n} b_i x_i \\ \text{subject to } \sum_{i=1}^{n} w_i x_i \leq C, \\ \qquad\qquad x_i \in \{0, 1\}, i = 1, 2, \cdots, n \end{cases}$$

where x_i is a binary variable equalling 1 if item i should be included in the knapsack and 0 otherwise.

The 0-1 KP has been given much attention in the literatures [11, 12]. It can be solved in a pseudo-polynomial time. The accurate solution can be obtained within one second e.g, using EXPBRANCH [12], for most data instances with up to 100000 items. A simple but very efficient heuristic algorithm for filling the knapsack is as follows: Ordering the items first according to their profit-to-weight ratio,

$$\frac{b_1}{w_1} \geq \frac{b_2}{w_2} \geq \cdots \geq \frac{b_n}{w_n}.$$

Then, in each step the item with the largest profit-to-weight ratio is packed into the knapsack if the item fits in the unused capacity of the knapsack, until the capacity is used up or no item fits for the residual capacity of the knapsack. This idea has been extended for solving multi-dimensional case, Many variations of the idea have been employed in many literatures [13], the item are selected according to $b_i / \sum_{j=1}^{m} \alpha_j w_{ji}$ for m constraints, where $\alpha_1, \alpha_2, \cdots, \alpha_m$ are given nonnegative weights. We develop a variation of the idea for the problem $\mathcal{P}$.

In the problem $\mathcal{P}$, the power saving p_i and hardware area a_i for block B_i are corresponding to the benefit b_i and the weight w_i of the item i, respectively, and the area constraint $\mathcal{A}$ corresponds to the capacity C of the knapsack problem. Compared with 0-1 knapsack problem, $\mathcal{P}$ has an extraordinary constraint: the time constraint $\mathcal{E}$. On the other hand, moving the block B_i to hardware must get the savings both in power and in execution time. Therefor, the problem $\mathcal{P}$ strongly correlated to the corresponding knapsack problem. This motivates us to construct an heuristic algorithm for solving the problem $\mathcal{P}$ by extending the

greedy idea described above with respect to the 0-1 knapsack problem. The proposed heuristic algorithm, denoted as HEU in this paper, is outlined as follows. To compress the paper, the formal description of HEU is omitted.

Algorithm HEU:

1. Sort the blocks $B_1, B_2, \cdots, B_n$ into nonincreasing order and then locate B_i at the r_i-th position, according to the ratio of the power-saving to area, *i.e.*, $\frac{p_i}{a_i}$, where $p_i = p_i^s - p_i^h$, $1 \leq i \leq n$. The smaller the r_i, the more power saving the block B_i has.
2. Sort the blocks $B_1, B_2, \cdots, B_n$ into nonincreasing order and then locate B_i at the t_i-th position, according to the ratio of the time-saving to area, *i.e.*, $\frac{e_i}{a_i}$, where $e_i = e_i^s - e_i^h$, $1 \leq i \leq n$. The smaller the t_i, the more time saving the block B_i has.
3. Assign $\alpha \cdot r_i + (1 - \alpha) \cdot t_i$ as the priority of block B_i for the given α, where $0 \leq \alpha \leq 1$ and $1 \leq i \leq n$.
4. Sort the blocks $B_1, B_2, \cdots, B_n$ into nondecreasing order according to the assigned priorities. The block with the lowest priority is first considered to be moved to hardware, as the smaller the priority, the more relative power saving and time saving the block B_i has.
5. Repeat the steps 3 and 4 for $\alpha = 0, 0.1, 0.2, \cdots, 1$. Set the solution of the largest power-saving to the optimal solution of $\mathcal{P}$.

There is a tradeoff between the power and the mixed constraints. In general, the larger the hardware area, the shorter the execution time is, while the higher the power becomes. Usually the given constraints are loose enough to provide large space of the feasible solutions. Otherwise, available hardware area hardly provides the solutions even if fit for the tight time constraint. For this case, the problem $\mathcal{P}$ is reduced to the 0-1 KP such as

$$\begin{cases} \text{maximize } \sum_{i=1}^{n} e_i x_i \\ \text{subject to } \sum_{i=1}^{n} a_i x_i \leq C, \\ \qquad\qquad x_i \in \{0, 1\}, i = 1, 2, \cdots, n \end{cases}$$

If its solution (obtained using EXPBRANCH in [12]) satisfies the time constraint $\mathcal{E}$, then the solution is a feasible solution of $\mathcal{P}$, otherwise, $\mathcal{P}$ has no solutions. Both are also the outputs of HEU if no solution can be found for each α in step 5.

The time complexity of the proposed heuristic algorithm is dominated by the sorting process for the data set of n elements, and thus bounded by $O(n \log n)$ [14]. To show the performance of the heuristic algorithm, we propose an exact algorithm for small-sized problems in the next subsection.

3.2 Exact Algorithm

Assuming that the optimal partitioning for $B_1, B_2, \cdots, B_{k-1}$ has been computed where the hardware area utilization is less than a, we now consider the method

to partitioning the blocks $B_1, B_2, \cdots, B_k$ within the available area a. This is achieved by first arriving at all partitioning possibilities based on representing the current block B_k in software or in hardware. The optimal partitioning results in the best possible power savings. If B_k is implemented in software, then B_k does not occupy hardware area while executes in e_k^s, and thus, the optimal partitioning for $B_1, B_2, \cdots, B_k$ for the hardware area a and execution time e is identical to the optimal partitioning for $B_1, B_2, \cdots, B_{k-1}$ for hardware area a and execution time $e - e_k^s$. If B_k is moved to hardware, the optimal partitioning for $B_1, B_2, \cdots, B_k$ can be found by examining partitioning for the blocks $B_1, B_2, \cdots, B_{k-1}$ for area $a - a_k$ and execution time $e - e_k^h$.

We employ the following notations to further describe our algorithm. $B(k, a, e)$ denotes the best power saving achievable by moving some or all the blocks from $B_1, B_2, \cdots, B_k$ to hardware of size a within the execution time e. The best power saving $B(k, a, e)$ equals $B(k-1, a, e-e_k^s)$ or the maximum between $B(k-1, a, e-e_k^s)$ and $B(k-1, a-a_k, e-e_k^h)+p_k$ as the block B_k is assigned either to software or to hardware respectively. $B(k, a, e) = B(k-1, a, e-e_k^s)$ corresponds to the case that hardware area is not enough for B_k, *i.e.*, $a_k > a$. Initially, $B(k, a, e)$ is set to $-\infty$ for all k and a if $e < 0$. Without loss of generality, the list of trial areas is set to $\langle 1, 2, \cdots, \mathcal{A}\rangle$ and the list of execution times is set to $\langle 1, 2, \cdots, \mathcal{E}\rangle$. Thus, the proposed algorithm DPP can be formulized as follows.

$$\begin{cases} B(k,a,e) = -\infty \quad for \;\; e < 0; \\ \\ B(k,a,e) = 0 \quad for \;\; k = 0, a = 0 \;\; or \;\; e = 0; \\ \\ B(k,a,e) = \begin{cases} B(k-1,a,e-e_k^s) & \textbf{if} \;\; a_k > a; \\ \\ \max \left\{ \begin{array}{l} B(k-1,a,e-e_k^s), \\ B(k-1,a-a_k,e-e_k^h)+p_k \end{array} \right\} & \textbf{else}; \end{cases} \\ \\ k = 1, 2, \cdots, n; \quad a = 1, 2, \cdots, \mathcal{A}; \quad e = 1, 2, \cdots, \mathcal{E}. \end{cases}$$

To compress the paper, we omit the formal description of the algorithm DPP. As an example, the instance shown by Fig.1 is calculated based on above formula. The optimal solution is $(1, 0, 1, 0)$ for the case that $\mathcal{A} = 5$ and $\mathcal{E} = 40$.

As each $B(k, a, e)$ can be calculated by one *addition* and one *max* between two data, the computing time for $B(k, a, e)$ is bounded by $O(1)$. This concludes that the time complexity of DPP is $O(n \cdot \mathcal{A} \cdot \mathcal{E})$ for n blocks, the list of trial areas $\langle 1, 2, \cdots, \mathcal{A}\rangle$ and the list of execution times $\langle 1, 2, \cdots, \mathcal{E}\rangle$.

4 Computational Results

Numerical computations such as addition and comparison dominate the proposed algorithm DPP. Without loss of generality, we test the correctness of DPP by using randomly-generated instances in C and run on a Pentium IV computer of 3GHz and 2GB RAM. In real application, in general, both the execution time and the the available hardware area are small values. Furthermore, the execution

time and power consumption in hardware are less than that in software for a same block. Hence, we use random data in $(0, 10)$ to simulate the execution time e_i^s and the area penalty a_i. After that, we randomly generate e_i^h in $(0, e_i^s)$. On the other hand, power consumption of each block hardly impacts the complexity of DPP, which is independent of the power. Without loss of generality, we set p_i^s, similarly to e_i^s, to a random data in $(0, 50)$ and then randomly generate p_i^h in $(0, p_i^s)$. We employ the following notations to further describe our computational results.

- P_sw (P_hw) denotes the power required in the case that all blocks are assigned to software (hardware).
- P_shw denotes the power required by the solution arrived at by our algorithm DPP.
- P_sav denotes the power saving by the partitioning solution obtained. It is calculated by the formula $P\_sav = (1 - \frac{P\_shw}{P\_sw}) \times 100\%$.
- The area constraint is set to $\mathcal{A}(\alpha)$ which is defined as $\mathcal{A}(\alpha) = \alpha \cdot sum\_area$, where $0 \leq \alpha \leq 1$ and sum_area denotes the hardware area required in the case that all blocks are assigned to hardware. In other words, the available hardware area is a fraction of sum_area.
- The time constraint is set to $\mathcal{E}(\beta)$ which is defined as $\mathcal{E}(\beta) = E\_hw + \beta \cdot (E\_sw - E\_hw)$, where $0 \leq \beta \leq 1$ and E_sw (E_hw) denotes the execution time in the case that all blocks are assigned to software (hardware). In other words, $\mathcal{E}(\beta)$ is bounded by E_hw and E_sw.

Table 1. Average errors (%) of solutions from the optimal solutions, 20 random instances with 200 blocks. $\delta = P\_sav$ of DPP $- P\_sav$ of HEU

	Power Saving P_sav (%) & error δ(%)								
Constraints	$\mathcal{E}(\frac{1}{3})$			$\mathcal{E}(\frac{1}{2})$			$\mathcal{E}(\frac{2}{3})$		
	DPP	HEU	δ	DPP	HEU	δ	DPP	HEU	δ
$\mathcal{A}(\frac{1}{8})$	-	-	-	-	-	-	18.76	17.88	0.88
$\mathcal{A}(\frac{2}{8})$	-	-	-	27.00	26.39	0.61	28.13	28.08	0.05
$\mathcal{A}(\frac{3}{8})$	31.47	30.60	0.87	34.76	34.68	0.08	34.81	34.76	0.05
$\mathcal{A}(\frac{4}{8})$	39.64	39.14	0.50	40.17	40.14	0.03	40.17	40.14	0.03
$\mathcal{A}(\frac{5}{8})$	44.33	44.30	0.03	44.34	44.31	0.03	44.34	44.31	0.03
$\mathcal{A}(\frac{6}{8})$	47.37	47.35	0.02	47.37	47.35	0.02	47.37	47.35	0.02
$\mathcal{A}(\frac{7}{8})$	49.29	49.27	0.02	49.28	49.27	0.01	49.28	49.27	0.01
$\mathcal{A}(1)$	50.18	50.18	0.00	50.18	50.18	0.00	50.18	50.18	0.00

Table 1 shows the power savings and the performance of the approximate solutions for different area constraints and the time constraints. $'-'$ denotes that there exist no solutions because of too tight constraints.

With the increase of the available hardware area, the power saving becomes higher and higher for both DPP and HEU under the same time constraint. For example, in the case of $\mathcal{E}(\frac{1}{2})$, the power saving of DPP is 34.76% for $\mathcal{A}(\frac{3}{8})$, while the power saving increases to 44.34% for $\mathcal{A}(\frac{5}{8})$. Similarly, under the same area constraint, the power saving also increase when the time constraint becomes loose. But the increase of power saving stops for looser area constraints. For example, in the case of $\mathcal{A}(\frac{4}{8})$, the power saving of DPP increases from 39.64% for $\mathcal{E}(\frac{1}{3})$ to 40.17% for $\mathcal{E}(\frac{1}{2})$, and then keeping the same for $\mathcal{E}(\frac{2}{3})$. This implies that some feasible solutions satisfied $\mathcal{E}(\frac{1}{2})$ become infeasible for the tighter constraint $\mathcal{E}(\frac{1}{3})$, the largest power saving for $\mathcal{A}(\frac{4}{8})$ has been found under $\mathcal{E}(\frac{1}{2})$, and thus $\mathcal{E}(\frac{2}{3})$ is correspondingly too loose.

The performance (power saving) of the approximate solutions produced by HEU are evaluated by δ calculated by $\delta = P\_sav\_of\_DPP - P\_sav\_of\_HEU$. In table 1, corresponding to the strongly correlated data instances of knapsack problem, which seems to be very hard to solve [11, 12], tighter constraints, *e.g.*, $\mathcal{A}(\frac{1}{8})$ & $\mathcal{E}(\frac{2}{3})$, $\mathcal{A}(\frac{2}{8})$ & $\mathcal{E}(\frac{1}{2})$ and $\mathcal{A}(\frac{3}{8})$ & $\mathcal{E}(\frac{1}{3})$, make the heuristic algorithm produce the approximate solutions of a little more errors, but bounded by 0.88%. δ becomes smaller with the increase of the available hardware area, *e.g.*, δ is reduced to 0.3% for $\mathcal{E}(\frac{2}{3})$ when $\mathcal{A}(\frac{1}{8})$ increases to $\mathcal{A}(\frac{4}{8})$. This corresponds to the weakly correlated data instances or unrelated data instances of knapsack problems, which are easier to solve [11, 12]. All errors in table 1 are so small that they will be ignored in most real-world problems. It is reasonable to believe that the proposed heuristic algorithm is applicable to the large problem sizes in the HW/SW partitioning.

5 Conclusions

Although several hardware/software partitioning techniques have been proposed over the last decade, they mainly focus on minimizing the execution time of the target system, where power consumption is ignored or appears as one of the constraints. In this paper, power consumption is emphasized and the HW/SW partitioning is considered with an objective to minimizing power under the constraints of hardware area and execution time. Based on the strong correlativity between the HW/SW partitioning and 0-1 knapsack problem, an efficient heuristic algorithm is proposed by extending the greedy idea of solving the 0-1 knapsack problem. The proposed heuristic algorithm runs in $O(n \log n)$, faster than the proposed exact algorithm, which is based on dynamic programming and running in $O(n \cdot \mathcal{A} \cdot \mathcal{E})$ for n code fragments under the hardware area constraint $\mathcal{A}$ and the time constraint $\mathcal{E}$. Computational results show the proposed heuristic algorithm can get nearly optimal solution for small-sized problems, and thus it is reasonable to believe that the proposed heuristic algorithm is also efficient for the large-sized problems of HW/SW partitioning.

References

1. R. Niemann and P. Marwedel, "Hardware/software partitioning using integer programming," in *Proc. of IEEE/ACM European Design Automation Conference (EDAC)*, 1996, pp. 473-479.
2. R. Ernst, J. Henkel, and T. Benner, "Hardware-Software Co-synthesis for Microcontrollers", *IEEE Design and Test of Computer*, 1993, vol. 10(4), pp. 64-75.
3. G. Quan, X. Hu and G. W. Greenwood. "Preference-driven hierarchical hardware/softwarepartitioning". In *Proc. of IEEE Int. conf. on Computer Design*, 1999, pp.652-657.
4. V. Srinivasan, S. Radhakrishnan, R. Vemuri. "Hardware Software Partitioning with Integrated Hardware Design Space Exploration". In *Proc. of DATE'98*. Paris, France, 1998, pp. 28-35.
5. R. Niemann and P. Marwedel. "An Algorithm for Hardware/Software Partitioning Using Mixed Integer Linear Programming". *Design Automation for Embedded Systems*, special Issue: Partitioning Methods for Embedded Systems. 1997, Vol. 2, No. 2, pp. 165-193.
6. M. Weinhardt. "Ingeger Programming for Partitioning in Software Oriented Codesign". *Lecture Notes of Computer Science 975*, 1995, pp. 227-234.
7. J. Henkel, R. Ernst. "An approach to automated hardware/software partitioning using a flexible granularity that is driven by high-level estimation techniques". *IEEE Trans. on VLSI Syst.*, 2001. vol. 9, no. 2, pp. 273-289.
8. A. Ray, W. Jigang, T. Srikanthan, "Knapsack model and algorithm for HW/SW partitioning problem", in *Proc. Of the Int. Conf. on Computational Science, Lecture Notes in Computer Science 3036*, 2004, pp.200-205.
9. J. Madsen, J. Grode, P. V. Knudsen, M. E. Petersen, and A. Haxthausen, "LYCOS: The Lyngby co-synthesis system," *Design Automation for Embedded Systems*, 1997, vol. 2, pp. 195-235.
10. S. A. Edwards, L. Lavagno, E. A. Lee, and A. Sangiovanni-Vincentelli, "Design of Embedded Systems: Formal Models Validation, and Synthesis," Proceedings of the IEEE, 1997, vol. 85, no. 3, pp. 366-390.
11. Martello, S. and P. Toth, "Knapsack Problems: Algorithms and Computer Implementations", John Wiley & Sons, 1990.
12. D. Pisinger, "Algorithms for knapsack problems", *Ph.D. Thesis, University of Copenhagen*, 1995.
13. A. Freville,"The multidimensional 0-1 knapsack problem: An overview", *European Journal of Operational Research*, 2004, vol.155, pp.1-21.
14. D.E. Knuth, "The art of computer Programming (Volumn 3), sorting and Searching(second Edition)", Addison-Wesley, 1998.

Exploring Design Space Using Transaction Level Models*

Youhui Zhang, Dong Liu, Yu Gu, and Dongsheng Wang

Microprocessor & SoC Research Center, Tsinghua Univ.
100084, Beijing, P.R.C
zyh02@tsinghua.edu.cn

Abstract. This paper presents THU-SOC, a new methodology and tool dedicated to explore design space by executing full-scale SW application code on the transaction level models of the SoC platform. The SoC platform supports alternative Transaction Level Models (TLMs), bus-functional model and bus-arbitration model, which enables it to cooperate with different levels of hardware descriptions. So, users are only required to provide functional descriptions to construct a whole cycle-accurate system simulation for a broad design space exploration in the architecture design. When the architecture is determined, the high-level descriptions can be replaced by RTL-level descriptions to accomplish the system verification, and the interface between the tool and the descriptions is unmodified. Moreover, THU-SOC integrates some behavior models of necessary components in a SoC system, such as ISS (Instruction-Set Simulator) simulator of CPU, interrupt controller, bus arbiter, memory controller, UART controller, so users can focus themselves on the design of the target component. The tool is written in C++ and supports the PLI (Programming Language Interface), therefore its performance is satisfying and different kinds of hardware description languages, such as System-C, Verilog, VHDL and so on, are supported.

1 Introduction

Rapid advancements in silicon technology have forced redefinition of a "system" - we have moved from systems-on-boards to systems-on-chip (SoC).

Because of the increasing complexity of such systems, the traditional RTL to layout design and verification flow proves inadequate for these multi-million gate systems [1]. There is a need for a hardware/software co-design approach that allows quick analysis of the trade-offs between implementation in hardware and software. And verification also becomes very important, so an efficient methodology is required to increase the number of reusable resources to improve the verification efficiency.

Moreover, although SoCs and board-level designs share the general trend toward using software for flexibility, the criticality of the software reuse problem is much worse with SoCs. The functions required of these embedded systems have increased markedly in complexity, and the number of functions is growing just as fast. Coupled with quickly changing design specifications, these trends have made it very difficult to predict development cycle time [2] [3].

* Supported by High Technology and Development Program of China (No. 2002AA1Z2103).

T. Srikanthan et al. (Eds.): ACSAC 2005, LNCS 3740, pp. 589 – 599, 2005.

THU-SOC is targeted to address the system-level design needs and the SW design needs for SoC design. It is a co-simulation & verification tool in C++ for SoC design. This tool adopts a bus-centric architecture. And different models of target hardware in a SoC system can be attached to the system bus through the programming interfaces of two Transaction-Level Models (TLMs), including bus-functional model and bus-arbitration model [4].

As we know, System level architects and application SW developers look for performance analysis and overall behavior of the system. They do not necessarily need and cannot make use of a cycle-accurate model of the SoC platform. However, a pure un-timed C model is not satisfactory either since some timing notions will still be required for performance analysis or power consumption. The capability of THU-SOC to bring HW/SW SoC modeling and characterization to the fore is the key.

THU-SOC can be decomposed into at least three major use paradigms:

- Creation of the SoC virtual platform for system analysis and architecture exploration.
- Use of the SoC virtual platform for SW development and system analysis by the systems houses SW or system designers.
- Use of the SoC platform for SW/HW co-verification.

Before system performance analysis, the designer can leverage the existing HW components delivered with THU-SOC to create the appropriate SoC platform architecture. The generic THU-SOC elements delivered include: one ISS simulator of CPU, an interrupt controller, a bus arbiter, a memory controller, a UART controller and the Linux OS. Especially, a Fast Decoding Simulation technology in our ISS is implemented. It can support varies of ISAs (Instruction Set Architectures) and provide an obvious improvement in decoding performance than currently known other similar technologies.

Then, using the provided programming interfaces of the SoC, system designers and application SW designers can use THU-SOC to simulate and analyze system performance in terms of latency, bus loading, memory accesses and tasks activity. These parameters allow for exploring system performance.

The next step during the design flow is to write high level descriptions of target hardware in C/C++ to integrate them to the tool through the interfaces of bus-arbitration model. Preliminary simulation can now be performed by executing the application code on the virtual platform to ensure that the code is functionally correct and the performance is satisfying.

After the architecture is determined, RTL descriptions based on bus-function model can replace the high level codes without any modification of the interface. Therefore, most resources can be reused to improve the verification efficiency.

As we know, in SoC design, hardware/software functional partition is a critical issue that determines the fate of the design. Therefore, it is necessary to construct system prototypes fast to evaluate their performance. The main value of our tool is to separates the communication details from the implementation to ease the exploration of the design space so that architects can focus on the design itself.

The remaining sections are organized as follows. Section 2 gives the relative researches. And the implementation details of our tool are presented in Section 3,

including the design philosophy, the internal architecture, and the main workflow between components. Section 4 introduces the result of its performance test and conclusions are given in Section 5.

2 Related Works

The concept of TLM first appears in system level language and modeling domain. [5] defines the concept of a channel, which enables separating communication from computation. It proposes four well-defined models at different abstraction levels in a top-down design flow. Some of these models can be classified as TLMs. However, the capabilities of TLMs are not explicitly emphasized. [6] broadly describes the TLM features based on the channel concept and presents some design examples.

TLMs can be used in top-down approaches such as proposed by SCE [7] that starts design from the system behavior representing the design's functionality, generates a system architecture from the behavior, and gradually reaches the implementation model by adding implementation details.

Some other research groups have applied TLMs in the design. [8] adopts TLMs to ease the development of embedded software. [9] defines a TLM with certain protocol details in a platform-based design, and uses it to integrate components at the transaction level. [10] implements co-simulation across-abstraction level using channels, which implies the usage of TLM.

Some other simulation tools have been released for design, too. For example, [11] presents a C/C++-based design environment for hardware/software co-verification. The approach is to use C/C++ to describe both hardware and software throughout the design flow. Other C/C++-based approaches to co-simulation include COWARE N2C [12] and PTOLEMY [13]. And [14] introduces a SystemC 2.0 TLM of the AMBA architecture developed by ARM, oriented to SOC platform architectures.

However, the transaction-level models (TLMs) are not well defined and the usage of TLMs in the existing design levels is not well coordinated. To solve this issue, [4] introduces a TLM taxonomy and compares the benefits of TLMs' use. Referring to [4], we employ two models, bus-arbitration model and bus-functional model, to enable our tool to cooperate hardware descriptions of different abstraction levels.

Compared with these existing systems, our tool provides more functions. It can cooperate with hardware descriptions of different abstraction levels with different TLMs, which can provide more simulation flexibility and higher simulation performance for users. Moreover, THU-SOC can boot Linux OS, which makes software development easier. At last, it supports different kinds of hardware description languages, such as System-C, Verilog, VHDL, through the PLI interface to implement SW/HW co-verification.

3 Implementation

3.1 The Philosophy

The principles of our design are presented as follows.

– High flexibility.

It is necessary to ensure that the tool can be used in different design phases. That is, it should cooperate with the behavior model and other more detailed models. So, we implement bus-functional model and bus-arbitration model to adapt different design levels.

– High performance.

Besides the bus models, some behavior models of necessary components in a SoC system, such as ISS simulator of CPU, interrupt controller, bus arbiter, memory controller, UART controller and so on, are provided, which speeds up the simulation and lighten the implementation burden of users. Moreover, we design a Fast Decoding Simulation technology in our ISS. It can support varies of ISAs and provide an obvious improvement in decoding performance than currently known other similar technologies.

– Easy to use.

Based on the pre-constructed components mentioned above, users can focus themselves on the high level description of the target component in the architecture design. Because the APIs of different bus models are identical, the high-level description can be replaced directly in the RTL design phase.

3.2 The Architecture

The simulated architecture contains processing elements (PEs), such as a custom hardware, a general-purpose processor, a DSP, or an IP, that are connected by the bus. Communication between processing elements is based on either message passing or global shared memory. In general, each PE has local memories as part of its micro architecture. If the local memory of a PE can be accessed from other PEs it becomes the global system memory.

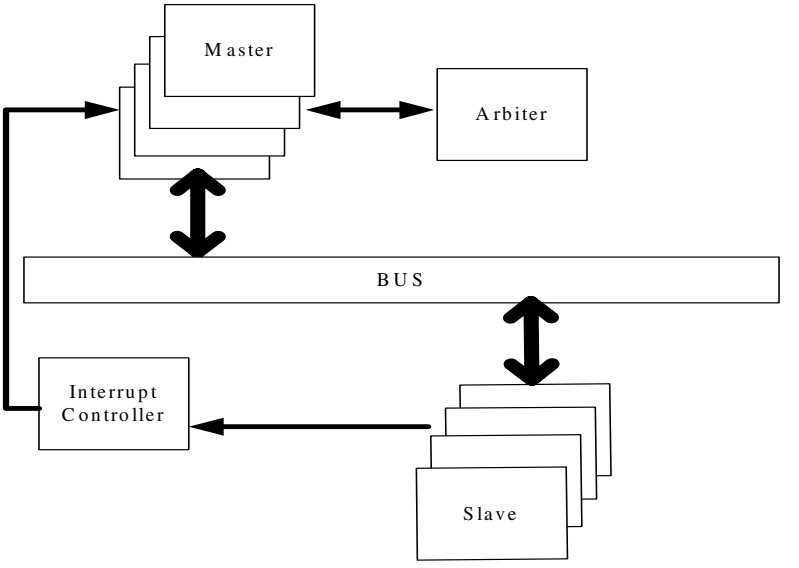


Fig. 1. The internal architecture

The PEs are decomposed of master and slave. The masters can initiate the transaction and read (write) data from (to) slaves. The slaves have memory-mapped registers which can be read and written by masters. In case of design with more than one slave on a bus, the system needs interrupt controllers. For design with more than one master on a bus, we need an arbiter to resolve multiple requests to the bus from masters. Figure 1 shows the typical architecture.

Of course, the key is how to implement the Transaction-Level Model of the system bus. In order to simplify the design process, designers generally use a number of intermediate models. The six intermediate models introduced in [4] have different design objectives. Since the models can be simulated and estimated, the result of each of these models can be independently validated. We employ two models from [4], bus-arbitration model and bus-functional model, which can be employed respectively in the design and verification phases.

In bus-arbitration model, the bus between PEs is called abstract bus channel. The channel implements data transfer while the bus protocol is simplified and neither of cycle-accurate and pin-accurate protocol details is specified. The abstract bus channel has estimated approximate time, which is specified in the channel by one or more wait statements per transaction.

In addition, we accomplish a bus-functional model, which can specify the transaction period in terms of the bus master's clock cycles. In this model, the abstract bus channel is replaced by the cycle-accurate and pin-accurate protocol channel. And wires of the bus are represented by instantiating corresponding variables and data is transferred following the cycle accurate protocol sequence. At its interface, a protocol channel provides functions for all abstraction bus transaction.

In order to speed up the simulation and enlarge the exploration space, we can perform architecture exploration with the bus-arbitration model, which has the approximate-timed communication. And then, after the PEs in design have been refined into more detailed descriptions that are pin-accurate and cycle-accurate, the bus-function model can be employed to verify the implementation.

3.3 API

The tool is implemented in C++, and some interface and class are defined to implement the channels. Two bus interfaces, MasterBusInterface and SlaveBusInterface, are declared. MasterBusInterface provides three main functions as follows:

MasterRequire: to acquire the control of the bus.
MasterFinished: to finish the current bus transaction and release the bus.
MasterWrite: to write to another PE.
MasterRead: to read from another PE.
GetAck: to get the acknowledge from the slave.
Similarly, the following functions are declared by SlaveBusInterface.
SlaveGetInfo: to get the current transaction info, including the target PE, read or write, address, data and so on.
SlaveRead: to read data from the bus.
SlaveWrite: to write data to the bus.

CBus class is defined to implement both of the interfaces. The class can be configured to behave as either of the two bus models and their APIs are the same.

In the run time only one instance of CBus is initiated, and all PEs will interact with each other through this instance. It means all PEs should maintain a pointer of the CBus instance. The class hierarchy is presented in Fig 2.

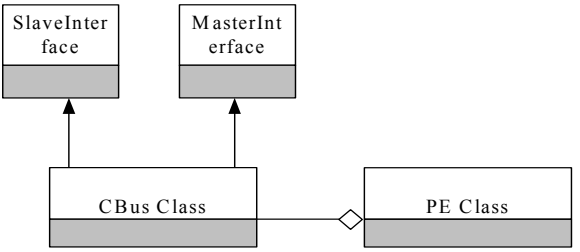


Fig. 2. The class hierarchy

3.4 Work Flow

For example, when CBus is configured to implement the abstract bus channel, a bus read transaction is processed as the following steps (Fig.3):

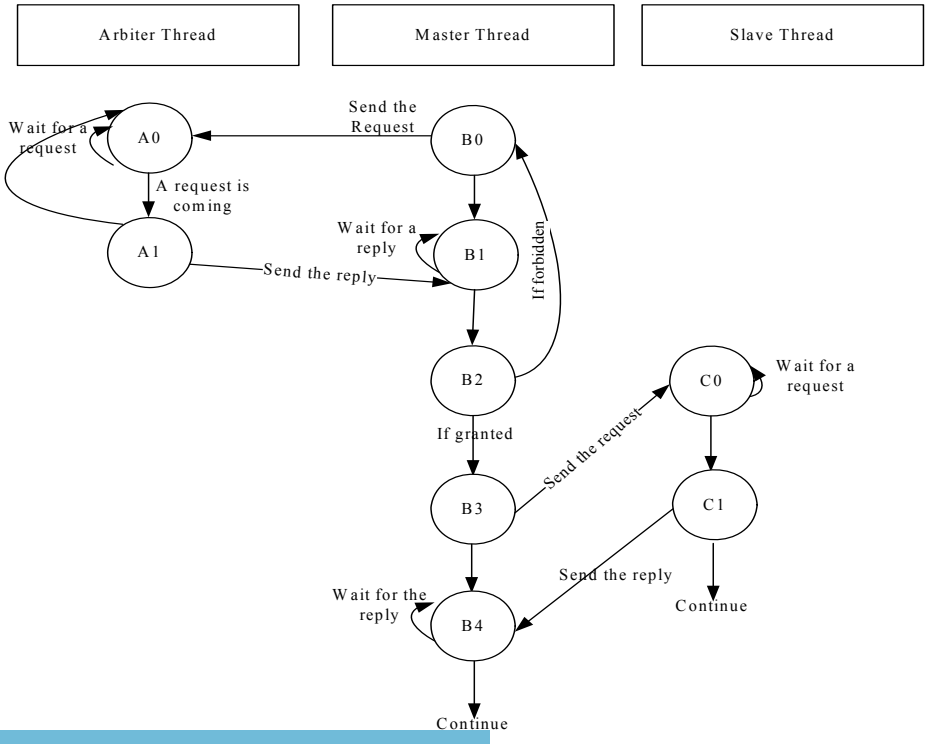


Fig. 3. Master and slave communication mechanism

1. The master PE asks the arbiter to get the permission to start a bus read through MasterRequire call.
2. When granted, the master sets the necessary info, such as the address, R/W command, to the CBus instance through MasterRead call. And then the master calls GetAck to check the valid reply.
3. At the same time other PEs will check the current bus transaction info through SlaveGetInfo call. Then, the target PE will handle the read request to provide the CBus instance with the wanted data. The user can insert zero or more wait statements before the acknowledgement to simulate approximate time.
4. Then, CBus will acknowledge the master.

Moreover, the cycle-accurate and pin-accurate protocol channel is also implemented by CBus. Now we provide a WISHBONE [15] bus function model, and more bus standards will be supported soon. Compared with the previous model, the transaction process is the same but each signal of the bus standard is instantiated and the detailed communication sequence is observed. More detailed info of WISHBONE can be referred to [15].

In our implementation, each master / arbiter PE owns an independent running thread and all slave PEs share a common thread. To keep the bus operation atomic, we adapt a round-robin running schedule. That is, each thread runs in turn. For example, if there are *n* master threads, one arbiter thread and one slave thread. All threads form the running queue and each thread will own one *1/(n+2)* time slice. When the CPU thread completes an instruction or begins a bus transaction, it will suspend itself and is appended to the end of the queue. Then, the next thread of the queue can get the running chance. After the latter completes its operation, the third thread will run and this process will go round and round. When the slave thread begins to execute, each slave will get the chance to check the current bus transaction info in turn, then the related slave can deal with the transaction. Several semaphore variables are employed to synchronize all threads.

3.5 Fast Decoding Technology in ISS

To suit different designs, a retargetable interpretative ISS is implemented to support different ISAs. As we know the interpretation-based simulation has higher flexibility and accuracy than the compilation-based, but its speed, especially the decoding speed, is slower. So, we design this technology to improve its decoding performance, and the testing result shows that it excels SimpleScalar [16] (interpretation-based), IS-CS [17] (static compilation and Just-in-Time compilation) and JIT-CCS [18] (Just-in-Time compilation and Translation Cache).

Compared with the existing optimizations that try to minimize the overhead of pleonastic decoding, our solution focuses on the improvement of the decoding algorithm itself.

We know mask codes and opcodes are used in most existing decoding algorithms. To identify one instruction, the algorithm complexity is O(n) and n is the amount of instructions.

The first step of optimization is to introduce a FSM. For one instruction Ai, its code is denoted as $a_{m-1}a_{m-2}\ldots a_1a_0$ and m is the code length. The following simple FSM (Fig.4) can be used to identify it and the complexity is O(m).

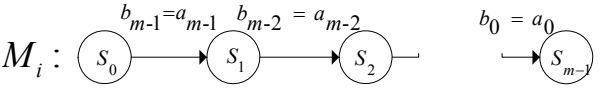


Fig. 4. The FSM to identify one instruction

For all n instructions, the FSMs can be combined into one. To simplify such a FSM is necessary to decrease the complexity. For example, the opcode (bit 31-26) and the function field in MIPS-II ISA determine the instruction type. So, only three or less judgments are needed to identify one instruction. But the most difficult is how to find out a general method to draw the coding characteristic from any given ISA. We design an algorithm to scan all instruction codes to draw the longest common sub-codes, and then generalize coding features to construct the decoding FSM automatically. The details are described as follows.

S denotes the target ISA, which contains *n* instructions and each is denoted by *Ai(i=1,…n)*. In addition, the current status of the FSM is *St*.

Step 1. Scan the code of *Ai* to find the first position of 0/1 transition, which is denoted by *Ki*.

Step 2. Get the minimum *K*, and do the following executions on the first *K* bits of all instructions:

a) If *K* bits of all instructions are 0, remove the beginning *K* bits from all instructions and the remaining ISA is denoted as *S'*. At the same time, the FSM status is transited to *St'*. For *St'*, this algorithm is executed again.

b) If exist some instructions whose beginning bits are all 1, find the shortest 1 sequence and the length is *L*. Classify all instructions based on different values of the beginning *L* bits and the result set is denoted as *Si*. *Si* contains some sub-sets and all codes in one sub-sets own the same beginning *L* bits. At the same time, the FSM status is transited to *Sti*. If the element number of one sub-set in *Si* is 1, this code is identified as an instruction. For other sub-sets, remove the beginning *L* bits from all instructions and this algorithm is executed again on the remaining.

Step 3. The end.

This algorithm is independent of the ISA. Owing to this algorithm, the retargetable ISS outperforms SimpleScalar, JIT-CSS and JS-IS simulators in the performance test where some Spec2000 benchmarks are used as the simulated applications.

4 The Usage

Now we are using this tool to design a Conditional Access Module (CAM) for Chinese DTV. For extensibility, it is a SoC chip that contains an embedded CPU, a memory controller, an INTC controller, the decryption component and so on. Embedded Linux is selected as the OS. Our design focus is the decryption component and most others are existing IPs.

To speed up the whole system verification and to enable software engineers to port OS as soon as possible, the co-verification and simulation tool is used fully in our project.

The design flow is described as follows:

1. Redesigning a MIPS instruction set simulator because our CPU core is MIPS-compatible.
2. Designing interfaces between the software and the decryption component.
3. Implementing the behavior description of the component to integrate to the tool on bus-arbitration model.
4. The whole simulation system is constructed now. Because approximate time can be estimated in bus-arbitration model and the behavior description is implemented relatively fast. The exploration of the design space is speeded up here. If the design target is satisfied, goto next step else goto Step 3. On the other side, software engineers can port the OS on the simulation.
5. Designing the pin-accurate RTL description in Verilog to integrate to the tool in bus-function model. In this phase, PLI is used as the programming interface, and the whole system can be simulated to verify its functionality. Moreover, testing traces generated in this phase are able to be compared with those from the high-level simulation directly to locate bugs if existing.

5 Performance Test

5.1 Fast Decoding

Our MIPS instruction set simulator implements all MIPS-II ISA instructions and simulates some system calls, so many Linux programs can run on this simulator. We choose *Adpcm* from Spec2000 as the testing program, which is a DSP benchmark to encode the G721 audio, and the PC equipped with 1.2GHz AMD Athlon processor and 512MB RAM is used as the test platform. This program is running respectively on Simple Scalar, JIT-CCS, JS-IS and our simulator and all results are recorded. In details, its speed is 4 times of SimpleScalar's, 1.5 times of JIT-CSS's and 15% higher than the speed of JS-IS when the latter's compilation time is neglected.

5.2 Bus-Arbitration Model and Bus-Function Model Performance

The compact version of Linux 2.4.28 for MIPS with one 4MB ram disk can boot up to the login interface on the SOC simulator, so we record the boot-up time respectively on the two models. On the bus-arbitration model with the embedded CPU ISS, the memory controller, the INTC controller and the UART controller, this time is about 130s. On the bus-function model, the time is about 980s. While the UART controller is replaced by the cycle-accurate verilog descriptions, this process lasts 6200s. In this test, Cadence NC-Verilog is used as the RTL running environment, which is integrated to our simulator through PLI. So, owing to the alternative TLMS, architect can explore the design space efficiently in the design phase.

6 Conclusions

In this paper we have described how we use C++ to construct a co-simulation & verification tool for SoC design. This will demonstrate the advantages of the usage of different TLMs in system design. Currently, it implements WISHBONE bus standard

and provides unified APIs for different levels of hardware description. Moreover, we design a Fast Decoding Simulation technology in the ISS of our tool. It can support varies of ISAs and provide the remarkable improvement in decoding performance than currently known other similar technologies.

We use this tool in the development of DTV CAM. It obviously speeds up the development and application engineers can devote themselves into the project in the beginning.

The next step is to implement more bus standards in bus-function model and more complex architecture, such as multiple buses containing the bus bridge.

Reference

1. Stan Y. Liao, Towards a new standard for system-level design. Proceedings of the eighth international workshop on Hardware/software Co-design. San Diego, California, United States. Pages: 2 - 6 . 2000.
2. Kanishka Lahiri, Anand Raghunathan, Sujit Dey, Fast Performance Analysis of Bus-Based System-On-Chip Communication Architectures. Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD),November 1999.
3. Frederic Doucet, Rajesh K. Gupta, Microelectronic System-on-Chip Modeling using Objects and their Relationships; in IEEE D&T of Computer 2000.Lukai Cai and Daniel Gajski. Transaction Level Modeling: An Overview. CODES+ISSS'03, October 1–3, 2003, Newport Beach, California, USA.
4. Lukai Cai , Daniel Gajski. Transaction level modeling: an overview. Proceedings of the 1st IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis. Newport Beach, CA, USA. Pages: 19 - 24. 2003.
5. D. Gajski et al. SpecC: Specification Language and Methodology. Kluwer, Jan 2000.
6. T. Grotker et al. System Design with SystemC. Kluwer, 2002.
7. S. Abdi et al. System-On-Chip Environment (SCE): Tutorial. Technical Report CECS-TR-02-28, UCI, Sept 2002.
8. S. Pasricha. Transaction Level Modelling of SoC with SystemC 2.0. In Synopsys User Group Conference, 2002.
9. P. Paulin et al. StepNP: A System-Level Exploration Platform for Network Processors. In IEEE Trans. On Design and Test, Nov-Dec 2002.
10. P. Gerin et al. Scalable and Flexible Cosimulation of SoC Designs with Heterogeneous Multi-Processor Target Architectures. In ASPDAC, 2001.
11. Luc Semeria, Abhijit Ghosh. Methodology for Hardware/Software Co-verification in C/C++. Asia and South Pacific Design Automation Conference 2000 (ASP-DAC'00), 01, 2000. Yokohama, Japan.
12. CoWare N2C, http://www.coware.com.
13. C. Passerone, L. Lavagno, M. Chiodo, A. Sangiovanni-Vintentelli, "Fast hardware/software co-simulation for virtual prototyping and trade-off analysis," Proceedings of the Design Automation Conference DAC'97, pp. 389-394, 1997
14. M.Caldari, M.Conti, etc. Transaction-Level Models for AMBA Bus Architecture Using SystemC 2.0. Proceedings of Design, Automation and Test in Europe Conference and Exhibition (DATE'03 Designers' Forum). 03, 2003. Munich, Germany.
15. www.opencores.org. WISHBONE System-on-Chip (SoC) Interconnection Architecture for Portable IP Cores. 2002.

16. Doug Burger, Todd M. Austin. The SimpleScalar Tool Set, Version 2.0. http://www.simplescalar.com.
17. Mehrdad Reshadi, Prabhat Mishra, Nikil Dutt. Instruction Set Compiled Simulation: A Technique for Fast and Flexible Instruction Set Simulation. Proceedings of the 40th conference on Design automation. Anaheim, CA, USA. 2003. Pages: 758 – 763.
18. Achim Nohl, Gunnar Braun, etc. A universal technique for fast and flexible instruction-set architecture simulation. Proceedings of the 39th conference on Design automation. 2002. Pages: 22 - 27.
19. System-C. http://www.systemc.org/.

Increasing Embedding Probabilities of RPRPs in RIN Based BIST

Dong-Sup Song and Sungho Kang

Dept. of Electrical and Electronic Engineering, Yonsei University,
134 Shinchon-Dong, Seodaemoon-Gu, 120-749 Seoul, Korea
dssong@dopey.yonsei.ac.kr, shkang@yonsei.ac.kr

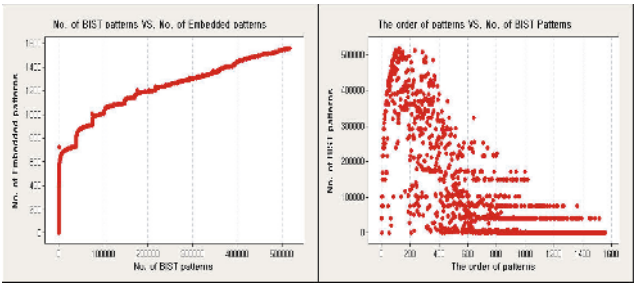
Abstract. In this paper, we propose a new clustered reconfigurable interconnect network (CRIN) BIST to improve the embedding probabilities of random-pattern-resistant-patterns. The proposed method uses a scan-cell reordering technique based on the signal probabilities of given test cubes and specific hardware blocks that increases the embedding probabilities of care bit clustered scan chain test cubes. We have developed a simulated annealing based algorithm that maximizes the embedding probabilities of scan chain test cubes to reorder scan cells, and an iterative algorithm for synthesizing the CRIN hardware. Experimental results demonstrate that the proposed CRIN BIST technique achieves complete fault coverage with lower storage requirement and shorter testing time in comparison with a previous method.

1 Introduction

Built-in self-test (BIST) for logic circuits implements most of the ATE functions on a chip, and it solves the problem of limited access to complex embedded cores in system-on-a-chip (SoC). The efficiency of a BIST implementation is determined by the test length and hardware overhead used to achieve complete or sufficiently high fault coverage. A pseudo-random test pattern generator or a linear feedback shift register (LFSR) has been widely adopted as a BIST pattern generator due to its low hardware overhead [1,2,3]. However, since there are many random pattern resistive faults (RPRFs) that can reduce the efficiency of the BIST, a large number of random patterns are required to achieve acceptably high fault coverage. To overcome the problem of RPRFs, techniques based on test point insertion [4,5], weighted random pattern [6,7], and mixed mode testing have been proposed, and these schemes offer various trade-offs between fault coverage, hardware overhead, performance degradation and test length.

Mixed mode techniques exploit a limited number of pseudorandom patterns to eliminate the easy-to-detect faults and a limited number of deterministic patterns to cover the remaining RPRFs. Unlike test point insertion, mixed mode techniques can reach complete fault coverage without imposing circuit modifications and causing performance degradation. Moreover, it is possible to obtain a trade-off between test data storage and test application time by varying the relative number of deterministic and pseudorandom patterns. Hence, a mixed mode

T. Srikanthan et al. (Eds.): ACSAC 2005, LNCS 3740, pp. 600–613, 2005.



(a) No. of BIST patterns VS. No. of Embedded patterns (b) The order of patterns VS. No. of BIST patterns

Fig. 1. Characteristics of RIN to embed test cubes for s9234 circuit

technique can be a good solution to the problem of achieving complete fault coverage and low testing time with relatively low storage and hardware overhead. Several studies on mixed mode BIST can be found in [8,9,10,11,12,13]. These include LFSR reseeding [8,9,10,11] and bit-flipping [12,13]. LFSR reseeding exploits multiple seeds for the LFSR, where each seed is an encoded deterministic test pattern. However, for circuits with a large number of RPRFs, this scheme has not only high testing time overhead required to feed seeds into the LFSR but also high hardware overhead resulting from the long LFSR where its size is determined by the number of care bits (0 or 1) in each deterministic test pattern. Bit-flipping alters certain bits in pseudorandom patterns to embed deterministic test cubes. This scheme achieves high fault coverage with practical test application times. However, since the number of bits required to be fixed is often too high, the logic overhead required may be considerable.

Recently, a reconfigurable interconnection network (RIN) BIST technique that can dynamically change the connections between the LFSR and the scan chains has been developed [14,15]. Fig. 1 shows the results of RIN to embed test cubes for s9234 circuit. In Fig. 1(b), 'The order of patterns' means the rank of each test cube where each test cube is sorted in descending order by the number of care bits. As shown in Fig. 1, RIN embeds two-thirds (1000) of the total test cubes (1557) within the first one-fifth (100000) of total applied BIST patterns (517733). And, most of the test cubes that are not covered in the first one-fifth of total BIST patterns belong to the first one-third of total test cubes. As a result, we can regard test cubes that have many care bits as random-pattern-resistant-patterns (RPRPs) which lengthen testing time and increase hardware overhead in RIN technique.

In this paper, we present a new clustered RIN (CRIN) BIST that can enhance the embedding probabilities of RPRPs. The proposed method improves the embedding probabilities of reformatted test cubes by using a scan-cell reordering technique based on the signal probabilities of deterministic test sets and specific hardware blocks. Experimental results for the ISCAS benchmark circuits demonstrate that the proposed method offers an attractive solution to the problem of achieving complete fault coverage and shorter testing time with relatively low storage and hardware overhead.

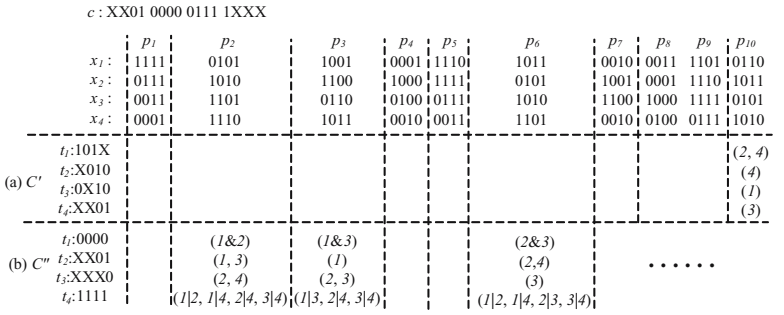


Fig. 2. An example of enhancing embedding probability

2 Proposed Logic BIST Scheme

2.1 Basic principles

The embedding probabilities of RPRPs with the LFSR generated test patterns are proportional to the number of Xs of each scan chain cube in the previous RIN BIST scheme. To increase the average number of Xs in the scan chain test cubes, the earlier RIN BIST scheme [14,15] uses a scan cell reordering method that scatters clustered care bits over all scan chains. However, if 0-care bits and 1-care bits are clustered into separate scan chains, the embedding probabilities of RPRPs can be improved.

An example of this enhancement when the LFSR generated patterns, P_1, P_2, $\cdots$, P_{10}, generate a XX01000001110XXX test cube C is illustrated in Fig. 2. In Fig. 2, (a) shows embedding procedures when care bits in the cube C are scattered over four scan chain cubes, t_1, $\cdots$, t_4, according to the earlier proposed scan cell reordering algorithm [14] and reformatted test cube C' can be embedded in the tenth LFSR generated pattern, P_{10}. The parenthesized number (2, 4) describes that the scan chain test cube t_1 can be generated if the second stage output x_1 or the fourth stage output x_4 of the LFSR is connected to the scan chain. In Fig. 2(b), the test cube C is transformed into C'' by respectively clustering the 0-care bits into the scan chain test cube t_1 and the 1-care bits into the scan chain test cube t_4. This is accomplished by applying the new scan cell reordering algorithm which will be presented in the next section. For convenience, we call the 0(1)-clustered scan chains AND(OR)-Chains and the remaining chains LFSR-Chains. Because care-bits in the test cubes are clustered into AND-Chains and OR-Chains, scan chain test cubes of LFSR-Chains that are fed by the LFSR have more X-values than the previous RIN, and thus the embedding probabilities of scan chain test cubes for LFSR-Chains is enhanced. In the proposed CRIN method, test patterns for AND(OR)-Chains are supplied by an AND(OR) Block which consists of LFSR-tapped 2-input AND(OR) gates. Because the outputs of the AND(OR) Block are weighted by 0(1), the embedding probabilities of 0(1)-clustered AND(OR)-Chains are further improved. In Fig. 2(b), 1&2(1|2) represents the bit-sequence that is generated by an AND(OR)

Table 1. Definitions of notations

Notation	Definition
m	The number of scan chains
L	The number of LFSR stages
l	Scan chain length
S_i	$0 \leq i \leq m \times l$, The order of scan cells
T_D	Deterministic test cube set
N_{TD}	The number of total test cubes
N_{AND}	The number of AND-Chains
N_{OR}	The number of OR-Chains
N_{LFSR}	The number of LFSR-Chains

gate whose two inputs are the first stage output and second stage output of the LFSR. In the above case, C'' can be covered with the LFSR generated test patterns P_2, P_3, P_6, and P_{10}. And, C'' has more candidate scan chain connections than the earlier RIN method. As a result, the proposed clustered RIN (CRIN) method solves the problem of achieving complete fault coverage and low testing time while maintaining relatively low hardware overhead.

2.2 New Scan Cell Reordering Algorithm

This section will explain the development of a new scan cell reordering algorithm based on the signal probabilities of given deterministic test cubes. This algorithm calculates the number of AND-Chains, OR-Chains, and LFSR-Chains while reordering scan cells to maximize the embedding probabilities of the reformatted scan chain test cubes. The proposed scan cell reordering algorithm is implemented by simulated annealing procedures and is described in Fig. 3. In this paper, we use the notations represented in Table 1.

(a) Calculate the signal probabilities P_i of each scan cell S_i using the first one-third of total test cubes where each test cube is sorted in descending order by the number of care bits. Let a deterministic test cube be C_j and a test cube set be $T_D = \{t_1, t_2, ..., t_{N_{TD}}\}$. The i-th bit of a deterministic test cube C_j and the signal probability of bit position i are defined as C_{ji}, and P_i, respectively. P_i is calculated by the following equation.

$$P_i = \frac{|\{C_j \in T_D | C_{ji} = 1\}|}{|\{C_j \in T_D | C_{ji} \neq X\}|} \quad for \;\; 1 \leq i \leq m \times l \;\; and \; 1 \leq j \leq \frac{N_{TD}}{3} \qquad (1)$$

The reason for including the first one-third of total test cubes into the computation of P_i is based on the experimental results that the most RPRPs reside in the one-third of total test cubes having the highest number of care bits.

(b) Reorder scan cells according to the P_i in an ascending manner. Also, scan cells that have P_i not exceeding 0.25 generate AND-Chains whereas scan cells that have P_i exceeding 0.75 generate OR-Chains. LFSR-Chains are

```
INPUT : Deterministic pattern set T_D, T_init , T_low , IPT, K_T, Initial scan cell order S_i
OUTPUT : Reordered scan cell S_i' , Reordered deterministic pattern set T_D'

Calculate signal probability of each scan cell S_i                    ..... (a)
Reorder scan cells and group AND-chains, OR-chains, and LFSR-chains   ..... (b)
For each group {
    T = T_init ;
    sv = Deterministic pattern set T_D corresponding current S_i ;
    Compute cost C(sv) ;                                              ..... (c)
    while (T > T_low) {
        if (no cost reduction in the last IPT iterations ) break ;
        for (i=0; i<IPT; i++) {
            (max, min ) = Find_swap_target(sv) ;                      ..... (d)
            (s_i', sv') = movement(max, min, sv, s_i) ;               ..... (e)
            Compute cost C(sv') ;
            ΔC = C(sv') - C(sv) ;
            if (ΔC < 0) sv = sv' ; s_i = s_i' ;
            else {
                p = e^-ΔC/T ;
                if (random(0,1) < p) sv = sv' ;
                else movement(max, min, sv', s_i') ;
            }
        }
        T = K_T x T ;
    }
}
T_D'= sv' ;
```

Fig. 3. New scan cell reordering algorithm

made up of the remaining scan cells. For the effect of step (b), 0(1)-care bits can be clustered in AND(OR)-Chains.

(c) After step (b), there are a large number of 0(1)-care bits and Xs in the rearranged scan chain test cubes of AND(OR)-Chains. However, since we are not concerned about the exact position of scan cell S_i in each group of scan chains (AND-Chains, LFSR-Chains, and OR-Chains), there is a possibility that the embedding probabilities of rearranged test cubes can be improved more and more. The highest embedding probabilities of CRIN BIST can be obtained when as many as possible Xs are included in the scan chain test cube. The following 'for-loop' performs an operation that scatters Xs over all scan chains. Note that the loop is applied to each group of scan chains. So, the exchange of two scan cells can take place in a group and the net signal probability of each group is not changed. Let the number of care bits in a j-th scan chain test cube of an i-th test cube be sp_{ij}. The cost function $c(sv)$ is calculated by the following equations.

$$ave_i = \frac{\sum_{j=1}^{m} sp_{ij}}{m} \quad for \ 1 \le i \le N_{TD} \tag{2}$$

$$dis_i = \frac{\sum_{j=1}^{m} (sp_{ij} - ave_i)^2}{m-1} \quad for \ 1 \le i \le N_{TD} \tag{3}$$

$$c(sv) = \frac{\sum_{i=1}^{N_{TD}} \sqrt{dis_i}}{N_{TD}} \tag{4}$$

(d) To reduce the cost of $c(sv)$, find two scan cells of which the position will be exchanged. Let a scan chain test cube that has the maximum value of $sp_{ij} - ave_i$ be $B_{max}(1 \leq i \leq N_{TD}, 1 \leq j \leq \alpha)$, and let a scan chain test cube that has the minimum value of $sp_{ij} - ave_i$ in the same test cube as B_{max} be B_{min}. Note that α is N_{AND}, N_{LFSR}, and N_{OR} for the AND-Chain group, the LFSR-Chain group, and the OR-Chain group respectively. The *maximum* is the bit position of a scan cell S_k corresponding to C_{ik}, where C_{ik} is any care bit in B_{max}, and the *minimum* is the bit position of a scan cell S_k corresponding to C_{ik}, where C_{ik} is any X-bit in B_{min}.

(e) Swap the positions of the two scan cells derived from step (d), and obtain the reorganized scan chain configurations S_i' and the reformatted deterministic test cube set T_D'.

2.3 Logic BIST Architecture

Fig. 4 describes the hardware architecture of the proposed logic BIST. The RIN blocks consist of multiplexer switches and they can be reconfigured by applying appropriate control bits to them through the inputs $D_0, D_1, \cdots, D_{g-1}$. The parameter g refers to the number of configurations used during a BIST session and it is determined using a simulation procedure. Test patterns for AND(OR)-Chains are supplied by the AND(OR) Block which consists of LFSR-tapped 2-input AND(OR) gates, and test patterns for LFSR-Chains are fed by the LFSR. Because the outputs of the AND(OR) Block are weighted by 0(1), the embedding probability of 0(1)-clustered AND(OR)-Chains can be improved. The control inputs $D_0, D_1, \cdots, D_{g-1}$ are provided by a d-to-g decoder, where $d = \log_2 g$. A d-bit configuration counter is used to cycle through all possible 2^d input combinations for the decoder. The configuration counter is triggered by the BIST pattern counter, which is preset for each configuration by the binary value corresponding to the number of test patterns for a given configuration.

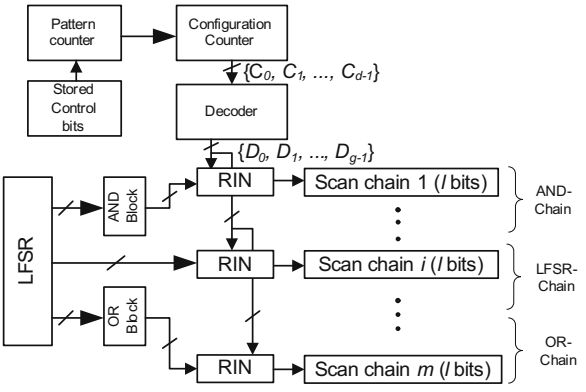


Fig. 4. Proposed logic of the BIST architecture

```
Set i = 1, j = 0 ;                                                    ..... (a)
while (T_D ≠ empty){
    Initialize connection configuration for each scan chain ;         ..... (b)
    while ( j < MaxSkipPattern +1 ) {
        Obtain the outputs of pattern generator for next l clock cycle ;   ..... (c)
        Find matched test cube with current connection configurations ;    ..... (d)
        if (match_success) {
            remove matched test cube ; reduce connection configuration ;   ..... (e)
            j = 0 ;
        }
        else j = j +1 ;
    }
    pattern simulation ;                                              ..... (g)
    i = i + 1 ;                                                       ..... (h)
}
```

Fig. 5. CRIN synthesis algorithm

3 CRIN Synthesis

To cover as many test cubes as possible for each configuration, RIN blocks connect the most suitable output stage of the AND-Block, the OR-Block, and the LFSR to AND-Chains, OR-Chains, and LFSR-Chains respectively, and this stage is determined by the iterative simulation procedure as represented in Fig. 5. We use M_{AND} and M_{OR} to denote the number of AND gates used in the AND Block and the number of OR gates used in the OR Block respectively. $ConnAND_x(i)(ConnLFSR_y(i), ConnOR_z(i))$ notation is also used to denote the set of output stages of the AND Block(LFSR, OR Block) that are connected to the scan chain $x(y, z)$ in configuration i. The simulation procedure is as follows.

(a) Set $i = 1$
(b) Set $ConnAND_x(i) = 1, 2, \cdots, M_{AND}$, $ConnLFSR_y(i) = 1, 2, \cdots, L$, and $ConnOR_z(i) = 1, 2, \cdots, M_{OR}$ $(0 \leq x \leq N_{AND}, N_{AND} < y < N_{AND} + N_{LFSR}, N_{AND} + N_{LFSR} \leq z \leq N_{AND} + N_{LFSR} + N_{OR})$. This means that each scan chain of AND-Chains, LFSR-Chains, and OR-Chains can be connected to any output stage of the AND-Block, the LFSR, and the OR-Block, respectively.
(c) Driving the LFSR for the next l clock cycle, we obtain M_{AND} l-bit vectors $\{O_{(AND)a}|a = 1, 2, \cdots, M_{AND}\}$, L l-bit vectors $\{O_{(LFSR)b}|b = 1, 2, \cdots, L\}$, and M_{OR} l-bit vectors $\{O_{(OR)c}|c = 1, 2, \cdots M_{OR}\}$. The $O_{(AND)a}$, $O_{(LFSR)b}$, and $O_{(OR)c}$ indicate the output steam of the a-th, b-th, c-th output stages of the AND-Block, the LFSR, and the OR-Block respectively for the l clock cycles.
(d) Find a test cube C^* in T_D that is compatible with the $O_{(AND)a}$, $O_{(LFSR)b}$, and $O_{(OR)c}$ under the current connection configuration $ConnAND_x(i)$, $ConnLFSR_y(i)$, and $ConnOR_z(i)$, respectively. C^* is reformatted for m

scan chains as a set of scan chain test cubes $\{t_1^*, t_2^*, \cdots, t_m^*\}$. The compatibility is valid when the following three conditions are satisfied.

1) when $1 \leq j \leq N_{AND}$:
   For all j, there exists $a \in ConnAND_j(i)$ such that t_j^* is covered by $O_{(AND)a}$.
2) when $N_{AND} < j < N_{AND} + N_{LFSR}$:
   For all j, there exists $b \in ConnLFSR_j(i)$ such that t_j^* is covered by $O_{(LFSR)b}$.
3) when $N_{AND} + N_{LFSR} < j \leq N_{AND} + N_{LFSR} + N_{OR}$:
   For all j, there exists $c \in ConnOR_j(i)$ such that t_j^* is covered by $O_{(OR)c}$.

(e) If no test cube is found in step (d), go to step (f) directly. Otherwise, remove the test cube C^* found in step (d) from T_D, and exclude the elements of the current connection configuration $ConnAND_j(i)$, $ConnLFSR_j(i)$, and $ConnOR_j(i)$ that cannot cover the t_j^* for all scan chains($1 \leq j \leq m$).

(f) If in the previous *MaxSkipPattern*+1 iterations, at least one test cube is found in step (d), then go to step (c). To limit the testing time, a *MaxSkipPattern* parameter which is defined as the largest number of pseudo-random patterns that are allowed between the matching of two deterministic test cubes is used.

(g) The simulation process for the current configuration i is concluded. Using test patterns generated during the current configuration i, remove C^* in T_D that can be embedded.

(h) Increase i by 1, and go to step (b). The iteration continues until the deterministic test cube set T_D is empty.

An example of the synthesis procedure is presented in Fig. 6. A circuit under test has six scan chains and the length of each scan chain is four bits. Pseudorandom patterns are generated by the LFSR which has a characteristic polynomial of $x^4 + x + 1$. The parameter *MaxSkipPattern* is set to 0. The AND-Block and the OR-Block have four AND gates and four OR gates respectively and inputs of each block are connected to the outputs of the LFSR with the configurations of (1, 4), (1, 3), (2, 4), (1, 2) where the parenthesized numbers indicate the stages of the LFSR, i.e. (1, 4) means that one input of an AND gate is connected to the first stage of the LFSR and the other input of the AND gate is connected to the fourth stage of the LFSR.

The original test cube set T_D and the reformatted test cube set T'_D which is calculated by the scan cell reordering algorithm introduced in section 2.2 are represented in Fig. 6. (A). The T'_D include six test cubes, C_1, C_2, $\cdots$, C_6 and each test cube C_i is divided into scan chain test cubes, $t_1, \cdots, t_4$ in Fig. 6. (B). In this example case, six scan chains consist of two AND-Chains, two LFSR-Chains, and two OR-Chains. In Fig 6. (C), the output of pattern generator is divided into patterns $p_i, i = 1, 2 \cdots$. Each pattern consists of 12 four-bit vectors where the first 4 four-bit vectors are outputs of the LFSR, the next 4 four-bit vectors are outputs of the AND-Block, and the remaining four-bit vectors are outputs of the OR-Block. The procedure that determines the connections between the pattern generator and each scan chain is shown in Fig. 6. (D). Step *Init* is

***(A) Scan cell reordering* :**

T_D	T_D'
C_1 : 11XXXXX01X0X0XXXXX10000X	C_1 : 00X00X0010XXX1XX1XXXXXX1
C_2 : XX11X01X10XXXXXXXX0XXX01	C_2 : 0XXXXXXXXX01X0101XXX11XX
C_3 : 01011XXXXXX1XXXXXXX00XXX	C_3 : XXX00XXX0XX0XX1XXXX1XX11
C_4 : 1XX0X01XXXXXX10X1XXXXXXX	C_4 : XXXXXXXX1X0X0X0XX11XX1XX
C_5 : XXXXX01X1XX10XXXXXXXX01	C_5 : 0XXXXX0XXX0XXXXX1XX111XX
C_6 : XXXXX01XX0X1XX1XXXXXXXX1	C_6 : XXXXXXXXXX0X1XX0XXX111XX

***(B) Scan chain test cubes* :**

		C_1	C_2	C_3	C_4	C_5	C_6
AND - Chains	t_1 :	00X0	0XXX	XXX0	XXXX	0XXX	XXXX
	t_2 :	0X00	XXXX	0XXX	XXXX	XX0X	XXXX
LFSR - Chains	t_3 :	10XX	XX01	0XX0	1X0X	XX0X	XX0X
	t_4 :	X1XX	X010	XX1X	0X0X	XXXX	1XX0
OR - Chains	t_5 :	1XXX	1XXX	XXX1	X11X	1XX1	XXX1
	t_6 :	XXX1	11XX	XX11	X1XX	11XX	11XX

***(C) Outputs of pattern generator* :**

			p_1	p_2	p_3	p_4	p_5	p_6
LFSR	1	x_1	0110	0100	0111	1010	1100	1000
	2	x_2	1011	0010	0011	1101	0110	0100
	3	x_3	0101	1001	0001	1110	1011	0010
	4	x_4	1010	1100	1000	1111	0101	1001
AND-Block	1	$x1$ & x_4	0010	0100	0000	1010	0100	1000
	2	$x1$ & x_3	0100	0000	0001	1010	1000	0000
	3	$x2$ & x_4	1010	0000	0000	1101	0100	0000
	4	$x1$ & x_2	0010	0000	0011	1000	0100	0000
OR-Block	1	$x_1 \mid x_4$	1110	1100	1111	1111	1101	1001
	2	$x_1 \mid x_3$	0111	1101	0111	1110	1111	1010
	3	$x_2 \mid x_4$	1011	1110	1011	1111	0111	1101
	4	$x_1 \mid x_2$	1111	0110	0111	1111	1110	1100

(D) Pattern matching procedure

	Init.)	*(a) p_1: C_1*	*(b) p_2 : C_4*	*(c) p_3 : None* *End of confi. 1* *Pattern simulation : None* *Init.*
$ConnAND_1(i)$:	(1, 2, 3, 4)	(1, 2, 3, 4)	(1, 2, 3, 4)	(1, 2, 3, 4)
$ConnAND_2(i)$:	(1, 2, 3, 4)	(2, 3, 4)	(2, 3, 4)	(1, 2, 3, 4)
$ConnLFSR_3(i)$:	(1, 2, 3, 4)	(2, 3)	(4)	(1, 2, 3, 4)
$ConnLFSR_4(i)$:	(1, 2, 3, 4)	(1, 3)	(1)	(1, 2, 3, 4)
$ConnOR_5(i)$:	(1, 2, 3, 4)	(1, 2)	(1)	(1, 2, 3, 4)
$ConnOR_6(i)$:	(1, 2, 3, 4)	(1, 2)	(1, 2)	(1, 2, 3, 4)

(d) p_4 : C_2	*(e) p_5 : None* *End of confi. 2* *Pattern simulation : C_5, C_6* *Init.*	*(f) p_6 : C_3*
(4)	(1, 2, 3, 4)	(1, 2, 3, 4)
(1, 2, 3, 4)	(1, 2, 3, 4)	(1, 2, 3, 4)
(2)	(1, 2, 3, 4)	(2, 3)
(1)	(1, 2, 3, 4)	(3)
(1, 2, 3, 4)	(1, 2, 3, 4)	(2)
(1, 2, 3)	(1, 2, 3, 4)	(2)

Fig. 6. An example of the synthesis procedure

the initialization step in which all connections $ConnAND_x(1)$, $ConnLFSR_y(1)$, $ConnOR_z(1)$ are set to (1, 2, 3, 4). Note that the (1, 2, 3, 4) of $ConnAND_x(1)$ and the (1, 2, 3, 4) of $ConnLFSR_y(1)$ have a different meaning. Since test patterns of AND-Chains are supplied by the AND-Block, the (1, 2, 3, 4) of

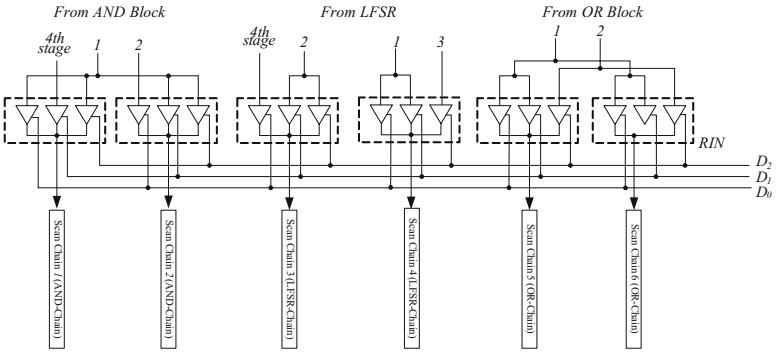


Fig. 7. An example CRIN configuration

$ConnAND_x(1)$ represents stages of the AND-Block. However, test patterns of LFSR-Chains are fed by the LFSR, and so the (1, 2, 3, 4) of $ConnLFSR_y(1)$ indicates stages of the LFSR. Similarly, the (1, 2, 3, 4) of $ConnOR_z(1)$ express stages of the OR-Block. In step (a), the first pattern p_1 is matched with the test cube C_1, and the possible connections are shown for each scan chain. In step (c), none of the cubes is compatible with p_3. Because the *MaxSkipPattern* is set to 0, the procedure for the current configuration is terminated. In step (d), which is the beginning of second configuration procedure, the candidate connection set is initialized with (1, 2, 3, 4) for each scan chain, and pattern p_4 is matched with the test cube C_2. Since the pattern p_5 is matched with none of the remaining test cubes, the procedure for the second configuration is terminated in step (e). Using the test patterns p_4 and p_5 which are generated in the second configuration, test cube C_5 and C_6 can be removed from T_D. The last test cube C_3 is matched with p_3 in the third configuration. Finally, the number of configurations for generating six test cubes is three and the number of test patterns needed is six. The connection configuration set for this CUT is $\{ConnAND_1(1), ConnAND_1(2), ConnAND_1(3)\} = \{1, 4, 1\}$, $\{ConnAND_2(1), ConnAND_2(2), ConnAND_2(3)\} = \{2, 1, 1\}$, $\{ConnLFSR_3(1), ConnLFSR_3(2), ConnLFSR_3(3)\} = \{4, 2, 2\}$, $\{ConnLFSR_4(1), ConnLFSR_4(2), ConnLFSR_4(3)\} = \{1, 1, 3\}$, $\{ConnOR_5(1), ConnOR_5(2), ConnOR_5(3)\} = \{1, 1, 2\}$, and $\{ConnOR_6(1), ConnOR_6(2), ConnOR_6(3)\} = \{1, 1, 2\}$ and Fig. 7 illustrates the connections between pattern generator and each scan chain.

4 Experimental Results

In this section, we demonstrate the efficiency of the proposed CRIN BIST for the ISCAS'89 benchmark circuits. During the scan cell reordering process proposed in section 2.2, we set up the parameters as $T_{init} = 5.0$, $T_{low} = 0.1$, $K_t = 0.97$, and $IPT = 500$. The LFSR has a characteristic polynomial of $x^{64} + x^4 + x^3 + x + 1$, and embeds randomly generated seed. The AND-Block and the OR-Block consist of 64 2-input AND gates and 64 2-input OR gates respectively and the

Table 2. Set of test cubes used in experiments (32 scan chains)

Circuit	No. of test cubes	Length of scan chain	No. of scan cells	No. of care bits
s5378	1285	7	214	12114
s9234	1557	8	247	19340
s13207	3221	22	700	23834
s15850	3257	20	611	27976
s35932	7477	56	1763	31392
s38417	7691	52	1664	87320
s38584	12216	46	1464	89090
Total				291066

selection of the two stage outputs of the LFSR which are connected to each gate is determined by the long distance first criterion. i.e., (1, 64), (1, 63), (2, 64), (1, 62), (2, 63), (3, 64) $\cdots$, where the parenthesized numbers represent stages of the LFSR. We calculated the gate equivalent value for the hardware overhead using the method suggested in [14], $0.5n$ for n-input NAND or NOR gate, and 0.5 for an inverter. We also chose 0.5 as the GE value for a transmission gate, and a GE value of 4 for a flip-flop. The set of test cubes used in experiments is obtained from Synopsys' TetraMax ATPG program without dynamic compaction, and by targeting all the single stuck-at faults. For experimental convenience, we added dummy scan cells to a CUT of which scan chains are unbalanced. So, all the ISCAS'89 circuits used in our experiment have balanced scan chains. ATPG results where each CUT contains 32 scan chains are presented in Table 2 and the number of care bits contained each test cube set are represented in the last column.

Table 3 presents the results of the previous RIN method and Table 4 shows the results for the proposed CRIN method where each CUT contains 32 scan

Table 3. Experimental results using the previous RIN [14,15] (32 scan chains, *MaxSkip-Pattern* = 5000)

Circuit	No. of Configurations	No. of BIST patterns	Storage requirement (bits)	Hardware overhead (percentage)	Encoding efficiency
s5378	16	257188	272	11.24%	44.54
s9234	32	517733	544	10.95%	35.55
s13207	10	205859	170	2.30%	140.20
s15850	36	588317	612	6.57%	45.71
s35932	4	27055	60	0.34%	523.20
s38417	243	2065818	4374	17.52%	19.96
s38584	20	518498	360	1.50%	247.47
Average	52	597210	913	7.20%	150.95

Table 4. Experimental results using the proposed CRIN (32 scan chains, *MaxSkipPattern* = 5000)

Circuit	No. of configurations	No. of BIST patterns	Scan chain groups	Storage requirement (bits)	Hardware overhead (percentage)	Encoding efficiency	Test time Reduction (percentage)	Storage requirement reduction (percentage)
s5378	11	186503	(11, 18, 3)	187	15.05%	64.78	27.48%	31.25%
s9234	16	296770	(12, 12, 7)	272	9.20%	71.10	42.68%	50.00%
s13207	6	157895	(18, 11, 3)	108	3.59%	220.69	23.30%	36.47%
s15850	25	356231	(16, 12, 4)	450	6.50%	62.17	39.45%	26.47%
s35932	3	19983	(17, 13, 2)	45	1.06%	697.60	26.14%	25.00%
s38417	133	1225964	(12, 17, 3)	2394	10.54%	36.47	40.65%	45.27%
s38584	8	265469	(9, 19, 4)	144	1.39%	618.68	48.80%	60.00%
Average	29	358402	(14, 15, 4)	514	6.76%	253.07	35.50%	39.21%

chains and the *MaxSkipPattern* parameter is set to 5000. The 'No. of configurations' column shows the total number of configurations needed to embed T_D. The total number of patterns applied to the CUT is listed in the 'No. of BIST patterns' column and this can be used as a measure of testing time needed to achieve 100% fault coverage. The parenthesized numbers in the 'Scan chain groups' column are the number of AND-Chains, LFSR-Chains, and OR-Chains, respectively, which were obtained from the proposed new scan cell reordering algorithm. The numbers in the 'Storage requirement' column are the amount of necessary storages which contain the information on the different number of patterns for each configuration. The encoding efficiency means the ratio of the number of care bits in the test set to the amount of storage needed. 'Test time reduction' is calculated by the ratio of the number of reduced patterns to the number of BIST patterns in the previous method and 'Storage requirement reduction' is calculated by the ratio of the reduced storage requirement to the earlier method's storage requirement. The significance of the differences between the experimental results for the previous work in Table 3 and the experimental results reported in [14,15] is limited by the fact that the test cubes, the LFSR's characteristic polynomial, and the LFSR's seed used in both experiments are not identical. The number of care bits in the test cubes used in our experiments is larger than the number of care bits in the test cubes for [14,15]. The total number of care bits in the test cubes used in our experiments is 291066 while the total number of care bits in the test cubes for [14,15] is 132812.

The efficiency of a BIST implementation is characterized by the test length and hardware overhead to achieve complete or sufficiently high fault coverage. The proposed CRIN method uses the scan-cell reordering technique and the specific hardware blocks that can improve the embedding probabilities of RPRPs. In consequence, the number of configurations needed to achieve 100% coverage of single stuck-at faults is reduced to 29 which is half that required by the previous system. The proposed CRIN BIST needs additional hardware blocks that consist

of M_{AND} AND and M_{OR} OR gates. Despite the need of additional blocks, the experimental results show that the hardware overhead is reduced from 7.2% to 6.75% by the effective reduction of total configurations. In comparison with the previous method, the proposed CRIN BIST technique reduces testing time by 35% and storage requirement by 39%.

5 Conclusion

The efficiency of logic BIST implementation is characterized by the test length and the hardware overhead required for achieving complete or sufficiently high fault coverage. This paper presents the clustered reconfigurable interconnect network BIST for the generation of deterministic test cubes. The proposed method uses a scan-cell reordering technique based on the signal probabilities of given test cubes and specific hardware blocks that improve the embedding probability of care-bit clustered test cubes. Though the proposed CRIN needs additional hardware blocks, experimental results demonstrate that the proposed approach requires less hardware overhead than the previous approach, which is a result of the reduction of total configurations. Experimental results also show that fewer control bits and shorter testing time are required compared to those needed for the previous approach. The proposed CRIN offers a viable solution to the problem of achieving complete fault coverage and low testing time while minimizing storage and hardware overhead.

Acknowledgments

This work was supported by the Ministry of Information & Communications, Korea, under the Information Technology Research Center (ITRC) Support Program.

References

1. P. H. Bardell, W. Mcanney, and J. Savir.: Built-in Test for VLSI: Pseudo-Random Technique. New York: Wiley, (1987)
2. V. D. Agrawal, C. R. Kime, and K. K. Saluja.: A tutorial on built-in self-test part 1: principles. IEEE Design & Test of Computers. volume: 10 (March 1993) 73–82
3. V. D. Agrawal, C. R. Kime, and K. K. Saluja.: A tutorial on built-in self-test part 1: applications. IEEE Design & Test of Computers. volume: 10 (June 1993) 69–77
4. A. J. Briers and K. A. E. Totton.: Random Pattern Testability by Fast Fault Simulation. Proc. of IEEE Int. Test Conf.. (1986) 274–281
5. Y. Savaria, M. Youssef, B. Kaminska, and M. Koudil.: Automatic Test Point Insertion for Pseudo-Random Testing. Proc. of IEEE Int. Symp. Circuit and Systems. (1991) 1960–1963
6. J. A. Waicukauski, E. Lindbloom, E. B. Eichelberger, and O. P. Forlenza.: A Method for Generating Weighted Random Patterns. IBM Journal of Research and Development. volume: 33 (March 1989) 149–161

7. H. S. Kim, J. K. Lee, and S. Kang.: A Heuristic for Multiple Weight Set Generation. Proc. of IEEE Int. Test Conf.. (2001) 513–514
8. S. Hellebrand, J. Rajski, S. Tarnick, and B. Courtois.: Built-in test for circuits with scan based on reseeding of multiple-poly-nomial linear feedback shift registers. IEEE Trans. Computers. volume: 44 (February 1995) 223–233
9. C. V. Krishna, A. Jas, and N. A. Touba.: Test vector encoding using partial LFSR reseeding. Proc. of IEEE Int. Test Conf.. (2001) 885–893
10. E. Kalligeros, X. Kavousianos, and D. Nikolos.: A ROMless LFSR reseeding scheme for scan-based BIST. Proc. of 11th Asian Test Symp.. (2002) 206–211
11. H. S. Kim, Y. J. Kim and S. Kang.: Test-Decompression Mechanism Using a Variable-Length Multiple-Polynomial LFSR. IEEE Trans. VLSI Systems. volume: 11 (August 2003) 687–690
12. H. J. Wunderlich and G. Kiefer.: Bit-flipping BIST. Proc. of IEEE/ACM Int. Conf. Computer-Aided Design. (1996) 337–343
13. G. Kiefer and H. J. Wunderlich.: Using BIST control for pattern generation. Proc. of IEEE Int. Test Conf.. (1997) 347–355
14. L. Li and K. Chakrabarty.: Deterministic BIST Based on a Reconfigurable Interconnection Network. Proc. of IEEE Int. Test Conf.. (2003) 460–496
15. L. Li and K. Chakrabarty.: Test Set Embedding for Deterministic BIST Using Reconfigurable Interconnect Network. IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems. volume: 23 (September 2004) 1289–1305

A Practical Test Scheduling Using Network-Based TAM in Network on Chip Architecture

Jin-Ho Ahn[1], Byung In Moon[2], and Sungho Kang[1]

[1] Dept. of Electrical & Electronic Eng., Yonsei University,
134 Shinchon-Dong, Seodaemoon-Gu, Seoul, 120-749, Korea
sominaby@soc.yonsei.ac.kr, shkang@yonsei.ac.kr
[2] School of Electrical Eng. & Computer Science,
Kyungpook National University,
1370 Sankyuk-dong, Buk-gu,
Daegu, 702-701, Korea
bihmoon@kpu.ac.kr

Abstract. It may be impractical to have TAM for test usage only in NoC because it causes enormous hardware overhead. Therefore, the reuse of on-chip networks for TAM is very attractive and logical. In network-based TAM, an effective test scheduling for built-in cores is also important to minimize the total test time. In this paper, we propose a new efficient test scheduling algorithm for NoC based on the reuse of on-chip networks. Experimental results using some ITC'02 benchmark circuits show the proposed algorithm can reduce the test time by about 5 - 20% compared to previous methods. Consequently, the proposed algorithm can be widely used due to its feasibility and practicality.

1 Introduction

While system performance needs to be increased exponentially to satisfy the desire of users, production costs should either remain the same or be reduced to ensure the competitiveness of the system. In this situation, SoC (System on Chip) is one of the more realistic and feasible solutions. Currently, SoC tries to raise the productivity of a system through IP (Intellectual Property), block designed beforehand or verified, reuses. It is expected that SoC including hundreds of PEs (Processing Element) and SEs (Storage Element) will appear in the near future by the development of new chip design and manufacturing technology. A data communication scheme between built-in cores in such a highly dense SoC will be a main design constraint in terms of system architecture, performance, robustness, power consumption, and cost. One of the new templates suitable for the architecture of a high-density SoC is the NoC (Networks on Chip) based platform [1].

NoC can be defined as a kind of SoC that has an interconnection architecture like micro-networks. Micro-networks, "on-chip networks" in other words, are an

T. Srikanthan et al. (Eds.): ACSAC 2005, LNCS 3740, pp. 614–624, 2005.

on-chip interconnection architecture that uses a network protocol based on communication layers. On-chip networks can provide many advantages on an NoC platform. First of all, On-chip networks are scalable and reconfigurable [2]. Thus it is easy to add or delete built-in cores on the network. Moreover, the operation clock of networks can be arbitrarily determined since on-chip networks do not require strict clock synchronization with embedded cores like computer networks. This is an important feature on NoC that includes multiple operation clocks.

The hardware structure of NoC consists of routers, cores, routing channels connecting between cores, and NIs (Network Interface) bridging between a core and a router [3]. The cores communicate with each other by sending and receiving packets com-posed of a header, a payload, and a trailer. Packet-based communication architectures can utilize the whole resources and bandwidth of networks effectively.

Like all other SoC, NoC has to be tested for manufacturing defects. NoC has nearly the same core test methodologies as SoC. However, some test architectures incorporating NoC characteristics have been proposed recently. Especially, TAM (Test Access Mechanism) is the most activated area in those architectures. TAM is the physical mechanism connecting cores from test sources or sinks, and it determines how efficiently test stimuli and test results can be transported. In earlier TAM architectures for SoC, an on-chip test bus has been the most efficient form for TAM. However, it is not feasible for NoC since the separate test bus causes excessive hardware overhead. Thus the reuse of on-chip networks for TAM becomes inevitable. An effective test scheduling is also important to minimize the total test time in NoC because NoC generally includes hundreds of cores.

In this paper, we propose a new efficient test scheduling algorithm for NoC based on the reuse of on-chip networks. The proposed algorithm has two dominant features. One is a deflection routing of test packets to satisfy simple router operations and minimize hardware overhead of routers. The deflection routing algorithm can route packets rapidly without buffering and explicit flow controls. Furthermore, we improved the performance of the algorithm by considering core priorities. The other feature is an asynchronous test clock strategy. As networks are normally much faster than embedded cores, several cores can be tested simultaneously. The asynchronous test clock platform can enhance test parallelization more than the multi-source/sink platform described in the previous algorithms.

First, we review prior works and present the purpose of our work in section 2. In section 3 and 4, two major features of the proposed test scheduling algorithm are presented respectively. The proposed test scheduling procedure is shown in section 5 with a pseudo-code. The experimental results using some ITC'02 benchmark circuits are given in section 6. Finally, this paper's conclusions are presented in section 7.

2 Related Work

The general concept of the reuse of on-chip networks for TAM is shown in [3]. Before the built-in core test, communication resources of NoC such as switches

or routers should be tested first. And then, we can advance the standard core test using the resources as TAM. Another approach on the subject of test architectures utilizing on-chip networks is proposed in [4]. The proposed network-oriented test architecture model is called NIMA (Novel Indirect and Modular Architecture), and contributes toward the basis for new test architecture that benefits from the reuse of NoC interconnect template. Test scheduling algorithms in NoC can be grouped roughly into two main categories: a packet-based scheduling and a core-based one. A core-based scheduling determines the test order of each core [5]. In this approach, the scheduler will assign each core a routing path, including an input port, an output port and corresponding channels that transport test vectors from the input to the core and the test response from the core to the output. Once the core is scheduled on this path, all re-sources on this path are reserved for the test of this core until the entire test is completed. Since the proposed idea maintains a pipeline from test vector input to test response output for a CUT (Core under Test), it shows fairly good results. However, it is impractical to use the core-based approach in a real situation because it is impossible to test cores with variable test clocks simultaneously. In other words, all NoC resources and CUTs should operate using the same clock during the test. A packet-based scheduling determines the order of generation and transmission of test packets for cores according to the priority of each core. E. Cota has proposed the test scheduling based on a packet-switching protocol [6]. In [6], test vectors and test responses per core are represented as a set of packets to be transmitted throughout the networks, and the packets are scheduled to minimize the total test time using test parallelism. Test parallelism means that several cores are tested simultaneously through maximizing the network bandwidth. Some enhanced versions of this algorithm have been reported. One is the supplement of power constraints [7]. Another reuses the embedded processors as test sources and sinks to increase test parallelism and reduce the test time [8]. Evidently, the packet-based scheduling is suitable for GALS (Globally Asynchronous and Locally Synchronous) architecture, and promises to fully exploit the characteristics of NoC. While a packet-based test scheduling having many merits, its experimental results have proven inferior to those of core-based one up to now.

3 Test Scheduling Using a Deflection Routing

A deflection routing, also known as a hot-potato routing, is based on the idea of delivering an input packet to an output channel in one cycle within a router. It assumes that each router has the equal number of input and output channels. In a deflection routing, when contention occurs and the desired output channel is not available, an input packet will pick any alternative available output channels to continue moving to the next router instead of waiting. Therefore, routers in the network do not have buffers to store the packets because input packets can always find a way to exit.

C. Busch [9] presents proofs regarding a hot-potato routing algorithm without explicit flow control under dynamic packet injection. In the algorithm, a packet

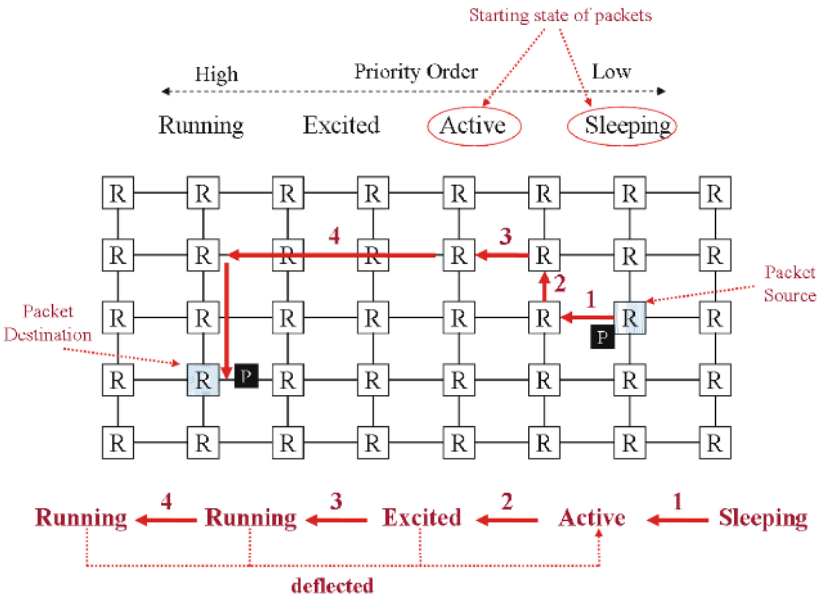


Fig. 1. Deflection Routing without Flow Control

always tries to follow any good channel. A good channel means one that brings it closer to its destination. In contrary, a bad channel is one that does not. If a packet cannot advance to its destination, the packet is forced to follow some bad channel, in which case we say the packet is deflected. When two or more packets are competing in the same router for the same output channel, we say that there is a conflict. In order to resolve conflicts, Busch makes use of packet priorities. There are four priority states in the algorithm: Sleeping, Active, Excited and Running. Sleeping is the lowest priority state and Running is the highest priority one. The higher priority packets are given routing precedence over the lower priority packets. Under dynamic analysis, the algorithm presented in [9] is shown to guarantee expected $O(n)$ delivery and injection times at $n * n$ mesh topology.

Figure 1 shows an example of the deflection routing in this paper for a packet. A packet P generated in a source follows a good channel, and its state will be changed from Sleeping to Active with some probability. In Figure 1, we assume the state change always occurs. If P in the Active state is deflected, P is changed into the Excited. The state of P in Excited will be Running if P can move closer to its destination by one step. P in the Running will be routed in its home-run path. A home-run path is defined as a path that only has one turn in it and follows the row first followed by the column. If P in the Excited or the Running state is deflected, its state will be changed into the Active.

A test scheduling for NoC should be comparable for as many NoC structures as possible. Therefore, the minimal additional logic and simple control for the test are quite helpful to make the scheduling algorithm robust in whatever NoC

will be used. The routing algorithm used for the proposed test scheduling in this paper is based on the Busch's algorithm that needs not to have any buffer in a router, and doesn't require any flow control. Thus the algorithm can be implemented regardless of the router structure and the routing algorithm used for data communication between cores. The only constraint is the same number of I/O channels such as mesh.

While most routing procedures implemented in the proposed test scheduling algorithm are similar to Busch's, test packets for each core can be assigned with a different priority state according to the core priority when the packets are generated in a test source. For example, a test packet that belongs to the core having the highest priority can start at a higher priority state than Sleeping. The priority of a core is determined by its test time. However, the determination of its start priority by core priorities has been somewhat heuristic until now. Currently, we assign the Active state to test packets of the core whose test time is over than a tenth of the total sum of test time of all embedded cores. This heuristic approach can reduce the total test time in parallel with a sorting of the cores in decreasing order of test time as introduced in previous studies.

4 Test Scheduling Using an Asynchronous Test Clock Platform

An asynchronous test clock strategy is another major feature in this paper. The test vectors and responses are transmitted via on-chip networks. We cannot fully take advantage of the merits of on-chip networks if we use the network as a dedicated routing path for a core test. On-chip networks are generally much faster than testing speeds of embedded cores. Therefore, several cores can be tested at the same time. A NI can compensate for the difference of operation clock frequency between the net-work and a core using buffers. In this approach, we need not to use the multi-source/sink platform described in the previous works. The difference of clock speed between the network and the core roughly corresponds to the number of test sources and sinks. For example, if on-chip networks operate two times faster than a tested core, it has the same effect as if there are two test sources and two sinks. However, an asynchronous test clock platform shows better results than multi-sources and sinks under the same condition since the asynchronous platform can fully schedule all cores by packets. The concept of an asynchronous test clock platform is presented in Figure 2. In Figure 2, R denotes a router, C is a core, and P is a test packet. The number attached to R, C, and P indicates their identity. If $R0$ operates by CLK_{R0} period, an input test packet for $C3$ at $R0$ is processed, and transmitted to a next router $R3$ for every CLK_{R0} time step. If CLK_{R0} is three times faster than CLK_{C3}, $R0$ can process test packets for $C1$ and $C2$, $P1$ and $P2$ in the Figure, while maintaining a pipelining of test vector of $C3$. At the present, Figure 2 only illustrates the basic idea. In the proposed algorithm, test packets for the same core are generated continuously at a test source until one test pattern is all generated if uninterrupted by higher priority cores.

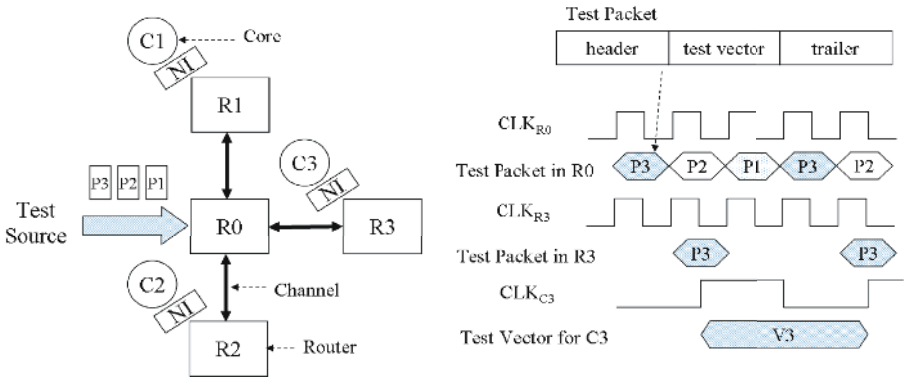


Fig. 2. Concept of Asynchronous Test Clock Platform

5 Test Scheduling Procedure

NoC can be characterized by several parameters such as topology, network protocol, structure and control of a router and a NI. In this study, we used 2-D mesh topology with channels set to be 32bit wide. A mesh structure is more practical and widespread application for NoC. Each node in a mesh is connected to its four neighbors via a bi-directional channel. A test source can inject packets at a rate of one packet per network time step, and a test sink can absorb packets at the same rate as the test source. Each packet contains a header including its destination, priority, and packet id indicating the position of the packet within a test pattern. While most previous studies have used a dimension-ordered routing based on a wormhole switching with a credit-based flow control, we adopted the deflection routing without a flow control as described in section 3. A router will receive a packet, and decide where to go with it using the proposed routing rules based on the packet's destination.

Before the proposed test scheduling algorithm starts, we should set the coordinates of routers, the priority of cores, the number of test patterns per core, the number of test packets per test pattern, the position of cores within topology, the position of a test source and sink, and the operation clock period of networks and cores. For convenience, we assumed all cores have the same test clock, and the network speed is deter-mined by multiples of the test clock. In the case of the priority of cores, a core with the longest test time has the highest priority. A test source and sink are directly connected to a router located in the boundary since we considered an external ATE case. For example, we illustrate the experimental conditions for d695 in Figure 3 and Table 1. Numbers at routers in Figure 3 are the coordinates of routers.

The algorithm begins with the injection of test packets of the highest priority core by the length of a test pattern. If it is done, a test source generates the test packets of the next priority core in order. However, the test source will hold the packet generation for a core whenever no output is available. Moreover, the test

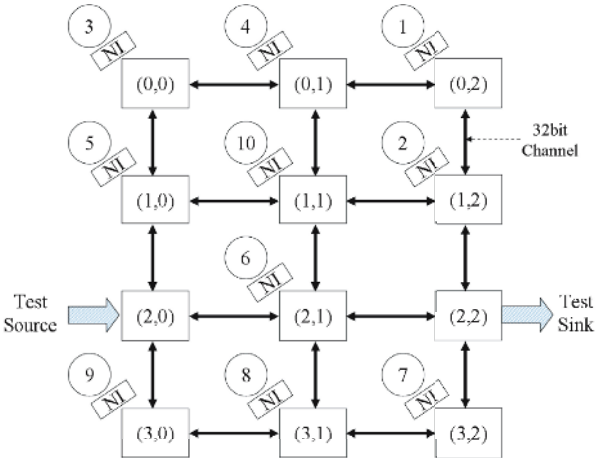


Fig. 3. NoC Topology in d695

Table 1. Test Packets for d695

Core	# of Test Patterns	# of Packets per Pattern	Test Time	Priority
1	12	1	25	9
2	73	7	588	8
3	75	32	2507	6
4	105	54	5829	2
5	110	55	6206	1
6	234	41	9869	0
7	95	34	3359	5
8	97	46	4605	3
9	12	64	836	7
10	68	55	3863	4

source immediately stops the current packet generation, and generates the test packets of the core having higher priority than the current one if the core is ready to receive the next packets. Incoming packets at routers will take priority over those generated/absorbed by the test source, sink, and cores. The time required to test a core is defined as follows.

$$T_{in} = \frac{T_{injection} + T_{router} \cdot N_{router} + T_{NI}}{S_{network}} \tag{1}$$

$$T_{out} = \frac{T_{absorption} + T_{router} \cdot N_{router} + T_{NI}}{S_{network}} \tag{2}$$

$$T_{packet} = T_{in} + T_{out}, T_{pattern} = \Sigma T_{packet}, T_{core} = \Sigma T_{pattern} \tag{3}$$

where T_{in} means the time to transmit a test vector packet from a test source to a core. T_{out} means the time to receive a test response packet from the core

```
program NoC_test_schedule
  set mesh size, the position of embedded cores;
  set the priority of cores by their test time;
  set test information of each core;
  set the position of a test source and sink;
  set the on-chip network speed;
begin
  until(all cores are tested) {
    repeat network speed {
      Router_operation in all routers;
      Test_source_operation;
      Test_sink_operation;
    }
    test operation in all cores;
  }
end

program Router_operation
begin
  while(there is an input packet) {
    case(packet.destination)
      when 'here':
        if(dest. is core and NI input port is idle)
          transfer packet into NI;
        else if(dest. is sink and sink input port is idle)
          transfer packet into sink;
        else
          deflects to random output port;
      when 'not here':
        deflection routing by the packet destination;
  }
end

program Test_source_operation
begin
  if(there is a free port) {
    fetch a test packet from queue of a test source;
    deflection routing by the packet destination;
    generate a next test packet and queuing
  }
end

program Test_sink_operation
begin
if(there is a input packet) {
    absorb a test packet;
    update test information;
    analyze the test response;
  }
end
```

Fig. 4. Pseudo-Code of NoC Test Scheduling

to a test sink. $T_{injection}$ is the time for injecting a packet in the test source, and $T_{absorption}$ is for absorbing in the test sink. T_{router} is the time for routing a packet in a router, and N_{router} is the number of routers in the routing path. T_{NI} is the time to process a packet in NI. $S_{network}$ is the relative clock speed of on-chip networks compared to a test clock, and is restricted to an integer value. In Equation (3), T_{packet} denotes the delivery time of one test packet from a test source to a test sink, $T_{pattern}$ denotes the time of one test pattern, and T_{core} is the final result that represents the total test time of a core. We assume $T_{injection}$, $T_{absorption}$, T_{router}, and T_{NI} are all 1. The pseudo-code of the algorithm is shown in Figure 4. The total test time of NoC equals to the summation of T_{core} of all embedded cores.

6 Experimental Results

The proposed algorithm is evaluated by the test application time through C-level simulations. The algorithm takes a minute in a d695, and is not over 10 minutes even in a p93791 on a Sun Microsystems 1.2-GHz UltraSPARC III. Though there are some variation of simulation results according to the position of a test source, a sink, and cores, the extent of variation due to it is not significant. Table 2 displays the simulation results of the proposed algorithm compared to previous results for four different ITC'02 benchmark circuits.

Table 2. Test Scheduling Results

Circuit Name	# of Sources/ Sinks or Relative Network Speed	Results in (5)	Results in (6)	Proposed
d695	2/2 or 2	18869	26012	20075
	3/3 or 3	13412	20753	12759
	4/4 or 4	10705	14785	10774
	- or 5	N.A	N.A	10088
	- or 6	N.A	N.A	10035
g1023	2/2 or 2	25062	31898	28616
	3/3 or 3	17925	22648	18434
	4/4 or 4	16489	18851	16168
	- or 5	N.A	N.A	15800
	- or 6	N.A	N.A	15360
p22810	2/2 or 2	271384	315708	312296
	3/3 or 3	180905	222432	208595
	4/4 or 4	150921	170999	162661
	- or 5	N.A	N.A	132982
	- or 6	N.A	N.A	115409
p93791	4/4 or 4	333091	435787	342819
	- or 5	N.A	N.A	285793
	- or 6	N.A	N.A	248108

Table 3. Statistics of Deflection Routing

Circuit Name	Mesh Size (H * V)	Relative Network Speed	# of Hops	# of Deflections
d695	3 * 4	2	10.4	5.1
		3	7.1	2.7
		4	8.8	4.1
		5	7.1	2.8
		6	8	3.5
g1023	4 * 4	2	11.5	5.3
		3	10.6	4.7
		4	10.3	4.5
		5	8.9	3.5
		6	9.6	4.1
p22810	6 * 6	2	26.6	15.8
		3	24.8	14.8
		4	25.1	15.2
		5	22.2	13.2
		6	20.4	11.9
p93791	6 * 6	4	24.2	14.5
		5	21.6	12.8
		6	21.6	12.9

Results in (5) based on the core-based scheduling is superior to those of the proposed in some cases. However, as mentioned in section 2, the core-based approach is so theoretical and impractical that it is not suitable for a real situation. As compared with results in (6) based on the packet-based scheduling similar to the proposed one, the test scheduling results using the proposed idea are quite noticeable in all cases. Furthermore, we can see from the results that the proposed algorithm is more effective under conditions of large circuits and high speed networks. This is very encouraging for the practicality and feasibility of the proposed algorithm.

In Table 3, the analysis results of the routing algorithm used in the proposed idea are discussed. The number of hops indicates the average number of router that a test packet visits from a test source to a test sink. The number of deflections indicates how often a test packet is deflected on the way from a test source to a test sink. Generally, the number of deflections is increased as circuit size grows since a greater percentage of packets have changed to higher states; thus, conflict within a router increases. However, the increment of network speed can reduce the confliction because the packet absorption rate increases linearly with respect to the network speed. Therefore, the average number of hops and deflections gradually decreases as the network clock speed increases.

7 Conclusions

In this paper, we propose a new efficient test scheduling algorithm for NoC based on the reuse of on-chip networks. The proposed algorithm adopts a deflection routing without a flow control to minimize test hardware overhead, and extends the routing algorithm to consider the core priority. Moreover, we develop an asynchronous test clock platform. The asynchronous test clock platform enhances the test parallelization more than the multi-source and sink platform described in the previous studies, thus improves test-scheduling results. Experimental results using some ITC'02 benchmark circuits show that the proposed algorithm provides superior results in spite of low hardware overhead. We expect the proposed test scheduling algorithm will be widely applicable due to its feasibility and practicality.

Acknowledgments

This work was supported by grant No. R01-2003-000-10150-0 from the Basic Research Program of the Korea Science & Engineering Foundation.

References

1. Benini, L., Micheli, G. D.: Networks on Chips: A New SoC Paradigm. IEEE Computer. (2002) 70–78,
2. Guerrier, P., Greiner, A.: A Generic Architecture for On-Chip Packet-Switched Interconnections. Proc. of IEEE DATE. (2000) 250–256
3. Vermeulen, B., Dielissen, J., Goossen, K., Ciordas, C.: Bringing Communication Networks on a Chip: Test and Verification Implications. IEEE Communications Magazine. (2003) 74–81
4. Nahvi, M., Ivanov, A.: Indirect Test Architecture for SoC Testing. IEEE Trans. on CAD. (2004) 1128–1142
5. Liu, C., Cota, E., Sharif, H., Pradhan, D. K.: Test Scheduling for Network-on-Chip with BIST and Precedence Constraints. Proc. of IEEE ITC. (2004) 1369–1378
6. Cota, E. et al: The Impact of NoC Reuse on the Testing of Core-based Systems. Proc. of IEEE VTS. (2003) 128–133
7. Cota, E., Carro, L., Wagner, F., Lubaszewski, M.: Power-Aware NoC Reuse on the Testing of Core-Based Systems. Proc. of IEEE ITC. (2003) 612–621
8. Amory, A. M. et al: Reducing Test Time with Processor Reuse in Network-on-Chip Based System. Proc. of the 17th Symposium on Integrated Circuits and Systems Design. (2004) 111–116
9. Busch, C., Herlihy, M., Wattenhofer, R.: Routing without Flow Control. Proc. of the 13th ACM Symposium on Parallel Algorithms and Architectures. (2001) 11-20

DRIL– A Flexible Architecture for Blowfish Encryption Using Dynamic Reconfiguration, Replication, Inner-Loop Pipelining, Loop Folding Techniques

T.S.B. Sudarshan, Rahil Abbas Mir, and S. Vijayalakshmi

Birla Institute of Technology and Science, Pilani , India
tsbs@bits-pilani.ac.in, rahilabbasmir@gmail.com
vijayalakshmi.seshadri@gmail.com

Abstract. Blowfish is used in a wide variety of applications, involving large amounts of data, demanding high-speed encryption, flexible key sizes and various encryption modes. Even though the ASIC implementations have better encryption speeds and throughput than FPGA ones, their flexibility is restricted. In blowfish, key generation and encryption stages function disjointedly rendering it suitable for dynamic reconfiguration. Fiestal network of the algorithm is better suited for inner loop pipelining and loop folding. Combining these architectural features with dynamic reconfiguration and replication results in a proficient architecture for blowfish described in this paper as the DRIL architecture. This four-tier architecture, involving both hardware and software designs, focuses on efficient hardware utilization, higher throughput, flexibility and better performance. Further, DRIL is a platform independent architecture. Performance evaluation on the XILINX SPARTAN 2E and VIRTEX 2 devices showed very high utilization of the FPGA device with a throughput of 259.615 x R Mbps on SPARTAN 2E and 515.218 x R Mbps on VIRTEX 2 devices, where R is the replication factor.

Keywords: Blowfish, Dynamic reconfiguration, replication, inner loop pipeline, loop folding, four –tier architecture, Platform independent architecture, DRIL Architecture

1 Introduction

Commerce and communication convergence towards computer networks has made cryptography as an indispensable tool for information technology. Blowfish is a simple, fast, compact, variably secure algorithm. Key feature of the blowfish algorithm are its immunity to attacks and its favorable structure for efficient hardware implementation [1, 2, 3, 4]. Blowfish is used in wide range of applications such as bulk encryption of data files [19], remote backup of hard disk [20]. Also multimedia applications use blowfish for encryption of voice and media files [21]. It is now being used in biometric identification and authentication, using voice, facial or fingerprint recognition [22]. Geographical information system uses blowfish for cryptographic protection of sensitive data [23]. These applications run in high-end servers, workstations, process bulk amount of data and demand high-speed encryption and higher throughput. Various computational solutions for these applications based on

T. Srikanthan et al. (Eds.): ACSAC 2005, LNCS 3740, pp. 625 – 639, 2005.

software and ASIC has been proposed earlier [7, 8, 9, 10, 11]. On one hand, software solutions are slow and insecure, while on the other hand, ASIC solutions that provides high-speed customized solution, lack flexibility. Solutions based on reconfigurable computing platform can offer both speed and flexibility. Flexibility in FPGA comes about due to two reasons – dynamic reconfiguration & replication**.** Blowfish algorithm has been implemented using various techniques such as, inner-loop pipeline and loop folding and has been shown to offer efficient utilization of resources and high throughput [12, 13]. We propose a reconfigurable architecture for blowfish algorithm that makes use of the above techniques of inner loop pipelining and loop folding and also for providing the flexibility and to enhance efficiency we propose two more techniques namely dynamic reconfiguration and replication. Based on these techniques a flexible architecture for blowfish algorithm called Dynamic reconfiguration, Replication, Inner loop pipeline, Loop folding Architecture abbreviated as DRIL is proposed in this paper. DRIL Architecture aims at efficient utilization of hardware through replication and loop folding, higher throughput through replication and inner loop pipeline, flexibility through dynamic reconfiguration and replication. The rest of the paper is organized in the following manner. Section 2 gives an overview of Blowfish algorithm. and the previous work, Section 3 proposes DRIL architecture and its design. Section 4 discuses the validation check, Section 5 gives the results and analysis and Section 6 gives the conclusion of the proposition.

2 Related Work

Blowfish Algorithm is a 64-bit block cipher using variable length key designed by Bruce Schneier [1,2,3]. The algorithm is executed in two steps, the first step handles the expansion of the key and next step does the encryption/decryption of the data. Key expansion converts the variable length key, which is between 32-bits and 448 bits into several sub-keys, totaling 4168 bytes. The first eighteen of these sub-keys form the P-box, the rest form the four S-boxes each having 256 keys. The data encryption is carried out in sixteen round Feistal network using these P and S – boxes. Each round consists of a key dependant permutation and data dependant substitution. Decryption is the same as encryption, except that the keys in the P-Box are used in the reverse order.

Various researchers have proposed ASIC implementations of the Blowfish Algorithm earlier. Different techniques has been exploited to achieve higher throughput such as, operator rescheduling and loop folding achieving maximum frequency of 50 MHz [10], pipelining the round of the Feistal network in blowfish into six stages achieving a maximum frequency of 66 MHz with a throughput of 266 Mbps [9]. For both these implementations the key generation were not implemented in hardware. Lai & Shu, proposed pipelining the round of the Feistal network in two stages with a key generation unit achieving a maximum frequency of 72 MHz [7,8]. Their hardware comprised of both the encryption and key generation units. Although full custom designs yield higher speeds and better performance, they do suffer from several disadvantages. They will be unable to respond to flaws discovered in the algorithm or to changes in standards. Additionally they cannot be optimized to suit

distinct situations. Reconfigurable hardware is a highly attractive solution for cryptographic algorithms to meet these problems. It combines the advantages of software with those of hardware implementations to offer algorithm agility, algorithm modifications, architectural efficiency and cost efficiency [5,6]. Several FPGA implementations of cryptographic block ciphers exploiting and exploring alternate design approaches are in literature [12, 13,14, 15, 16, 17, 18]. Although the best-known implementations of Blowfish algorithm reported in the literature have been unable to achieve comparable throughput. Chodowiec et. al. proposed Blowfish implementation on Altera FPGA achieving a maximum frequency of 10MHz and throughput of 40 Mbps [12]. Singpiel et al. have proposed an implementation in FPGA Coprocessor micro Enable increasing the speed almost by a factor of 10 [11]. To the best of the authors knowledge, there has been no architecture proposed for dynamic reconfiguration and replication for Blowfish algorithm We explore various architectural options and propose DRIL, a flexible architecture with efficient hardware utilization, better performance and higher throughput when compared to existing architectures.

3 DRIL Architecture

Blowfish Algorithm has features that facilitate the use of many architectural options. In addition to using techniques like loop folding, inner-loop pipelining and on-chip P boxes and S boxes, DRIL architecture uses dynamic reconfiguration and replication. Moreover it is flexible enough to be used efficiently and fit seamlessly in different system designs. The principles used in the DRIL architecture are dynamic reconfiguration, replication, inner-loop pipelining, loop folding and P & S boxes.

Runtime reconfiguration or dynamic reconfiguration allows customization of circuits for the problem at hand. They offer considerable speed and area advantages especially to cryptographic algorithms. This allows dynamic circuit specialization based on specific key and mode [5, 6]. Blowfish algorithm involves key generation computation that is done only once for a particular key when the algorithm is initiated. Key generation need not be present when the data is being encrypted or decrypted. The key generation unit, needed only for the initial sub key generation, can be dynamically loaded when required. After key generation this hardware can be dynamically reconfigured to an encryption block/decryption block. This methodology saves on resources by using the same available hardware for encryption and key expansion implementation.

Replication is a method by which the hardware is replicated as many times as possible in the FPGA device to increase the throughput. Replicating the encryption algorithm hardware ‘R’ times on the FPGA device increases the throughput by ‘R’ times at most. Depending upon the needs and the availability of the hardware resources on FPGA device, the same blowfish implementation can be replicated as many times for encryption/decryption. Replication provides the blowfish implementation the flexibility required to adapt to various needs of different system designs where it is used. Inner loop pipelining is a sub-pipelined architecture resulting in an increase in throughput by pipelining one round of blowfish. This pipeline has five stages. The data has to iterate through one round sixteen times. Because of

pipelining now five distinct data blocks would iterate simultaneously instead of one block increasing the throughput. [12, 13].

Inner loop pipelining is a sub-pipelined architecture resulting in an increase in throughput by pipelining one round of blowfish. This pipeline has five stages. The data has to iterate through one round sixteen times. Because of pipelining now five distinct data blocks would iterate simultaneously instead of one block increasing the throughput. [12, 13].

Loop folding avoids the use of excess hardware by implementing only one round of Feistal network over which the cipher iterates. Only one out of the sixteen rounds in the Fiestal network of blowfish are implemented saving on hardware resources. The data is made to flow through this hardware sixteen times for the sixteen rounds of blowfish. For the blowfish algorithm it also means a less complex P and S –box access because these boxes are accessed in only one round. [12, 13].Loop folding reduces the hardware complexity associated with the access of the P and S - boxes besides using minimum hardware. Reduction in the complexity means a reduction in the hardware implementation. Contemporary FPGA's include embedded blocks of memory where data as large as 3- Megabits can be stored. Together these factors make it possible to have on-chip P and S - boxes. The on chip P and S - boxes have very small memory access times in comparison to the implementation where the P and S boxes are on a separate memory chip**.**

3.1 Blowfish Architecture

The principle operations in blowfish architecture are the iteration of data over the same round sixteen times and the retrieval of encrypted/decrypted data or sub keys. The hardware is divided into five components to carry out these operations. The five components are the Inner loop pipelined unit, the control unit, on chip P and S-boxes, the decoder unit and Input and Output buffers.

INNER PIPELINE UNIT: This unit implements one round of blowfish shown in fig 3.1. For the encryption of a data block, the data block is iterated sixteen times through the inner loop pipelined unit. The pipelined unit has five stages the first of which is the P-stage, the second, third and fourth are the S-box stages and the last is S box- XOR stage. The P-box stage implements the XOR of data with the P-box values. The three S-box stages perform the retrieval, addition and XOR operations of the data with values from S1, S2 and S3 boxes. The last stage performs the addition of the data to the value from S4 box but if this is the sixteenth iteration then the left and right halves of data are also XORED with the seventeenth and eighteenth values of the P box.

CONTROL UNIT: This unit keeps track of the number of iterations done so far and the stage in which each data block is present. Since the Key generation iterates a fixed 521 times to generate the P and S boxes, this unit also tracks this number. Ring counters are used for this purpose.

ON-CHIP P BOXES AND S BOXES: The on chip P and S boxes are implemented as 32 – bit registers. The P box comprises eighteen of these registers storing the eighteen values of the P box. Each of the S boxes has 256 registers to store the 256 S-box values.

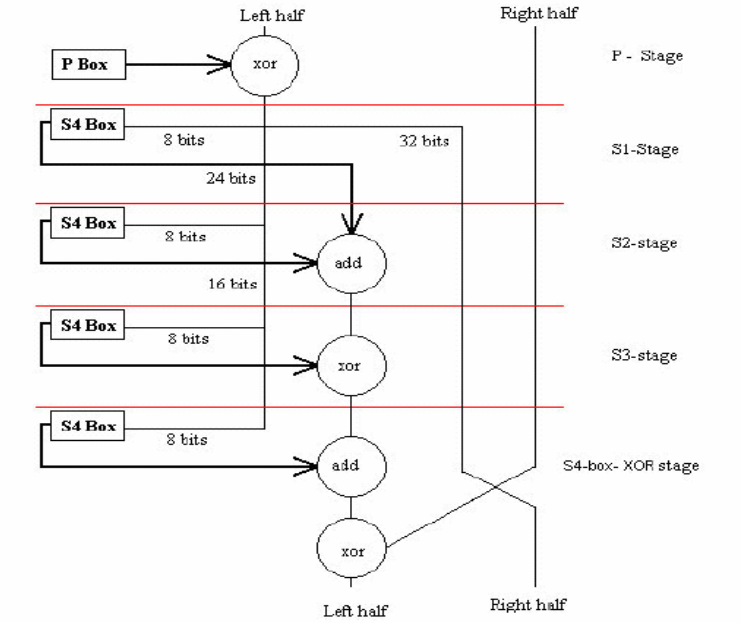


Fig. 3.1. Division of one stage of Fiestal Network into five stages of the pipeline

DECODER UNIT: The decoder unit takes the input from each of the stages, and retrieves the correct P or S –box value. This unit is different for key generation and encryption/decryption units of the hardware.

INPUT AND OUTPUT BUFFERS: These buffers hold the data before and after the encryption. Both types of buffers contain five registers. All the registers of the input buffer can be written simultaneously. The pipeline is fed from the input buffer by shifting out values from the Input buffer to the pipeline, one register at a time. The values in the pipeline after encryption are shifted out to the output buffer, one register at a time. All the registers of the output buffer can be read simultaneously

3.2 Key Generation Unit

The key generation unit shown in fig 3.2 generates sub-keys. This unit comprises of four elements inner loop pipeline unit, control unit, decoder unit, on chip P and S boxes. Standard values XORED with private key are in on–chip P and S-boxes, when the device is configured for key generation. Starting with the null vector, the inner-loop pipelined unit encrypts its own output using the values from on–chip P and S-boxes. The entries in these boxes are replaced with these output values sequentially.

Decoder unit addresses the on–chip P and S-boxes. The indexed values to the decoder unit are provided by the second, third, fourth and fifth stages of the inner-loop pipelined unit. Generation of a sub-key through the sixteen iterations of the data in the inner-loop pipelined unit replaces two entries in the on–Chip P and S -boxes. Control unit tracks these iterations and the entries that are to be replaced. Control unit

makes provisions (signals) to inform the external interface about the generated output and for the replacement of the correct entry in the on–chip P and S –boxes. The key generation unit generates exactly 521 output values. The generation of the new output is dependent on previous, so the pipeline will process one block at a time even though it can process five. This is the reason why a single decoder is used to address the four S boxes.

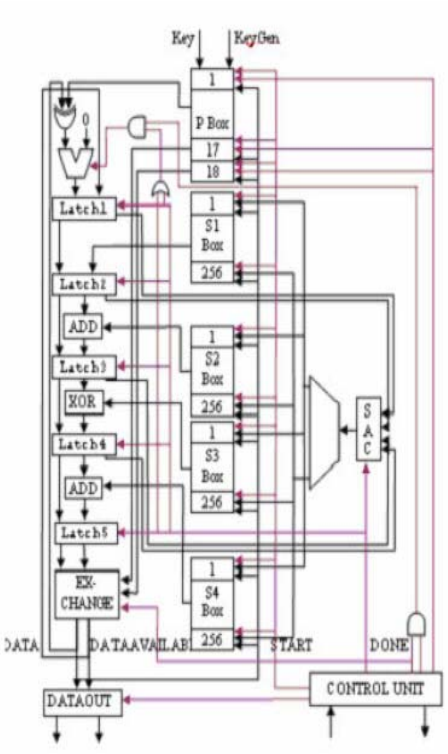


Fig. 3.2. Key Generation Unit

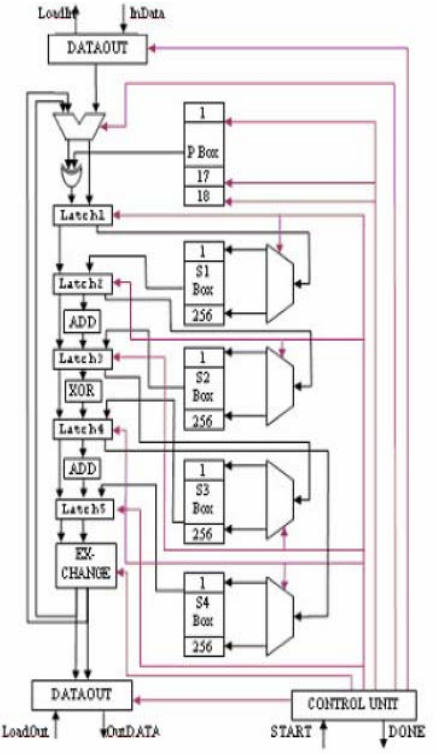


Fig. 3.3. Encoder Decoder Unit

3.3 Encoder and Decoder Units

This unit shown in fig 3.3 encrypts data. Besides the four elements used by the key generation unit, the encryption unit also includes input and output buffers. The input buffer stores the values that are fed to the inner-loop pipelined unit. Inner-loop pipelined unit encrypts these data through sixteen iterations using the values from the on–chip P and S -boxes. These boxes have the sub-keys generated by the key generation unit. On–chip P and S -boxes are addressed by the decoder unit. The indexes from the second, third, fourth and fifth stages of inner-loop pipelined unit are fed simultaneously to the decoder unit, which uses these indexes to address the four S – boxes concurrently. After 16 iterations, the encrypted values are placed in the output buffers. External interface can retrieve the data from the output buffers or place new data in the input buffers at any time during the sixteen iterations of encryption, when inner-loop pipelined unit is not accessing these buffers. Control unit informs the

external interface when they can access the buffers. It also tracks the iterations in inner-loop pipelined unit and indexes the on chip P boxes in straight order during encryption and in reverse order during decryption. Five blocks are processed by each of the five stages of pipeline simultaneously.

3.4 DRIL Architecture Design

For a proper functioning, the applications should be aware of the features of the underlying device, where the hardware is configured. Applications have to interact with the hardware by sending and receiving data and other control information during encryption. DRIL is a platform independent architecture. Here applications need not be aware of the architecture at any time. Blowfish architecture adapts to suit the needs of applications for which it will do the encryption.

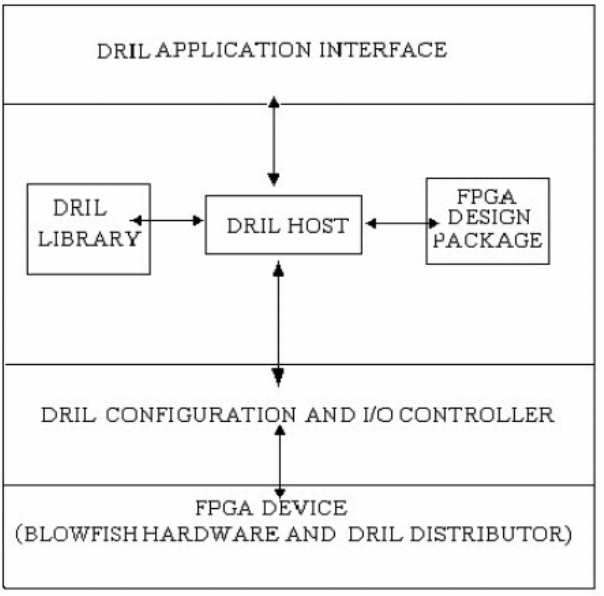


Fig. 3.4. DRIL Architecture

DRIL is a flexible design. It can identify whether the application may already be using a reconfigurable device. Accordingly, it can change the number of replications in blowfish hardware so that it can fit with application hardware on the FPGA device. The issue here would be a tradeoff between the speed of encryption desired by the user and the area occupied on device. The hardware is efficiently utilized by DRIL architecture. ASIC implementations of encryption algorithms occupy fixed areas but DRIL changes the number of encryption replications to best fit the entire FPGA. DRIL, apart being flexible and hardware efficient, is high-speed encryption-decryption architecture. Once configured, the architecture on the device is dedicated to encryption or decryption solely. It has all the advantages of the blowfish algorithm.

It uses variably length session key, is fast and simple. DRIL has two parts onto itself, the hardware design and the software design. The four-tier DRIL architecture consists of DRIL user interface, DRIL host, DRIL configuration I/O controller and a dynamically reconfigurable device. The four-tier DRIL architecture is shown in figure 3.4.

3.4.1 DRIL Software

DRIL APPLICATION INTERFACE: DRIL Application interface interacts with user/application and obtains information about the file that is to be encrypted or decrypted, the session key, the replication number and whether encryption or decryption operation is to be performed on the file. The session key is obtained only once for every session. If the session key is provided then the key-generation unit of the blowfish architecture is invoked to generate new sub-keys. DRIL can replicate the encryption-decryption unit to increase throughput. The application can control the number of replications by specifying the replication number. However, the number of replications is fixed for a session. The maximum number of replications is dependant on the area supported by the reconfigurable device.

DRIL HOST: This is the central controller that manages and synchronizes the function of all the different components in the architecture. It uses the design package for the FPGA device, the library files and the session key to generate the configuration file for the key-generation. It instructs the configuration and I/O controller to download the configuration file into FPGA. It performs similar operations for the encryption-decryption unit. It calculates the maximum number of replication possible for FPGA device during setup. It controls the data input to the FPGA and also collects the data from the FPGA with the help of Configuration and I/O Controller.

DRIL LIBRARY: DRIL library has source files that hold the information about the key generation and encryption unit. It also consists of the interfacing files for communication with the external *device specific* packages and the controller. P and S box files containing the standard values are also present in the library. These files are used by DRIL HOST to generate the configuration file for the key generation unit, encryption unit and distributor unit.

DRIL CONFIGURATION AND I/O CONTROLLER: The configuration and I/O controller configures the FPGA to function as key generation unit or as an encryption-decryption unit. This unit will read the sub-keys from the FPGA during the key generation. During encryption it is going to partition the data and feed it to the FPGA. The input of data to the FPGA should be in a manner so that none of the replicated units on the FPGA is waiting. Waiting would mean wastage of the encryption cycles and loss in throughput. The same unit also collects the data from the FPGA replications and merges it. Configuration and I/O controller exchanges handshake signals to synchronize the flow of data with FPGA device. Moreover it also performs the scan operations to check that the device is up and functioning correctly.

DRIL Hardware

DRIL DISTRIBUTOR: The encryption-decryption unit is replicated on the FPGA device. The throughput increases by the number of replications only if all of them are

functioning simultaneously. For this purpose these replications need to be fed data simultaneously. But that would mean using a large interface between the Configuration and I/O controller and the FPGA device. The FPGA device may not be able to support such a large interface. The distributor holds the responsibility to distribute the data between the replications, thereby making the interface small. It takes in data, and replicated hardware Id and feeds the replication with the data. It encapsulates the operation in such a way that it is hidden to the upper layers.

DRIL HARDWARE: The FPGA device and the blowfish architecture together with the distributor form the DRIL Hardware. The FPGA device is configured for encryption-decryption and key-generation unit. It is under the control of the configuration and I/O controller that configures and reconfigures it for the two stages and also provide the input and collect the output from this device using the distributor.

3.5 DRIL Operation

The application layer of the DRIL architecture communicates with the applications that need to encrypt the data. The application layer provides the information supplied by the lower layers, about the availability and operations status of the FPGA device, to the applications. It gets information about the data to be encrypted or decrypted, the destination folder, the session keys, the type of operation performed and the number of replications from applications. This information is passed on to the DRIL host. DRIL host operations are of two parts- Session-Start mode and in-Session mode, which gets reflected down the hierarchy. During the in-session mode, DRIL host will encrypt or decrypt the data. It will read the data from the file and provide this data to the Configuration & I/O controller. The Configuration & I/O controller in this mode would simply supply the FPGA device with the data and Wait for encryption or decryption. It returns back the encrypted or decrypted data to the DRIL host, which writes it back to the user specified file. The simplicity of this mode follows directly from the blowfish algorithm and is the reason for the fast encryption of data.

The start session is more complicated. It is indicated by a change in the session key. On chip P and S boxes of the encryption-decryption unit are changed whenever the session key changes. The DRIL host first generates the new P and S-box files with the new key. It retrieves the appropriate files from the DRIL Library and then merges them, invokes the external FPGA packages and generates the Key generation unit configuration file. This file is passed on to the Configuration and I/O controller, which configures the FPGA with this unit and retrieves the new sub –keys from this unit. These sub keys are returned to the DRIL host. The DRIL host also has prior information about the number of replications that can be supported by the FGPA device that is obtained from configuration and I/O controller at the startup. Using this information and with the help of DRIL library, it generates the distributor and the encryption /decryption unit configuration file. This file is passed on to the Configuration and I/O controller, which configures the FPGA with the encryption and decryption unit. This unit will do the encryption only for one session, so long as the key does not change.

4 Design Validation of DRIL Architecture

Validations being essential to establish the correct functioning of the blowfish architecture and the integrity of DRIL architecture, two types of validation were carried out for DRIL Architecture. The first validation was the functional verification of the blowfish architecture. The individual stage of the Feistal Network and the entire blowfish hardware were independently verified. Blowfish hardware were implemented in Verilog HDL and functionally verified with ModelSim Verilog simulator. The results obtained from the design code were compared and validated with the standard test vectors and with the results from the Bruce schenier code. The generation of sub keys, correct cipher text for the given plain text and correct plain text for the cipher text was verified.

DRIL architecture involves many components like application interface, DRIL host, DRIL configuration and I/O controller, DRIL distributor and blowfish hardware that form the different layers in the design. These components communicate and function as a group for the encryption and decryption of data. The second validation is performed to verify the functioning of the DRIL architecture. The DRIL application interfaces were implemented in JAVA–AWT. It is a graphical user interface for the validation purpose. Configuration and I/O controller were implemented in Verilog PLI and DRIL host in JAVA. The DRIL library consists of the file required for key generation, encryption and decryption units.

The application user interface takes in the session key, the source file, destination file and the required number of replication. In the start session mode, the DRIL host generates the Key generation unit. The configuration and I/O controller send the start signal to initiate the key generation unit. The keys are generated in 521 cycles by the key generation unit. Every time a key is generated the configuration and I/O controller is informed to collect the key. Next the Host generates the Encryption-Decryption unit. Depending on the number of replications the Encryption-Decryption unit and the distributor units are generated. The host starts the In-session mode. The configuration and I/O controller send the start signal to initiate the encryption unit. It also partitions the data into 64 bit blocks and sends it to replication sub units. After the data is encrypted it is collected back by the configuration and I/O controller and given to the DRIL host. DRIL host writes the data in the destination file specified by the user. The area the encryption unit occupies and the total area available on the FPGA device decide replication factor. The areas occupied by the encryption unit on different platforms were determined. SPARTAN 2E and VIRTEX 2 were chosen as platforms due to their support for embedded distributed RAM and Block RAM. The validated design were synthesized and implemented on SPARTAN 2E devices of 300K and 600K gates and VIRTEX –2 devices of 500K, 1000K and 1500K gates with a speed grade of –6.

5 Experimental Results

The lowest layer in the DRIL architecture is the blowfish hardware implemented on different FPGA device. Encryption, Decryption, Key Generation of the data are responsibilities of the lower layer. The performance of the DRIL architecture is directly dependant on the performance of this layer. DRIL architecture efficiently

utilizes the underlying device. The results obtained are tabulated in Appendix. Table 1 provides the device utilization of the different devices with a replication factor of 1, 2 and 3. The device utilization is high for the devices with smaller total area with a replication factor of one and it is high for devices with large total area with a replication factor of three. Table 1 also shows the IOB utilization of the device. The advantage of this design is that the IOB utilization remains same and does not increase as many times as the replication factor because of the design of DRIL distributor. The flexibility provided in the DRIL architecture is that the replication factor can change depending on platform, it is able adapt and utilize the resources efficiently. The frequency and throughput yield for various replications on different FPGA devices are shown in Table 2. The maximum operational frequency of the hardware for different replication but on the same device is constant. If the resources are available in the device, as indicated under the FIT column, replication and inner-loop pipelining increases the throughput although the maximum operational frequency remains same. Maximum throughput is the throughput when the pipeline is assumed to behave ideally and is calculated by applying equation (1).

$$\textit{Throughput Max} = (\textit{block size} * \textit{clock frequency} * \#\ \textit{pipeline stages}) / (\textit{encryption clock cycles}) \quad (1)$$

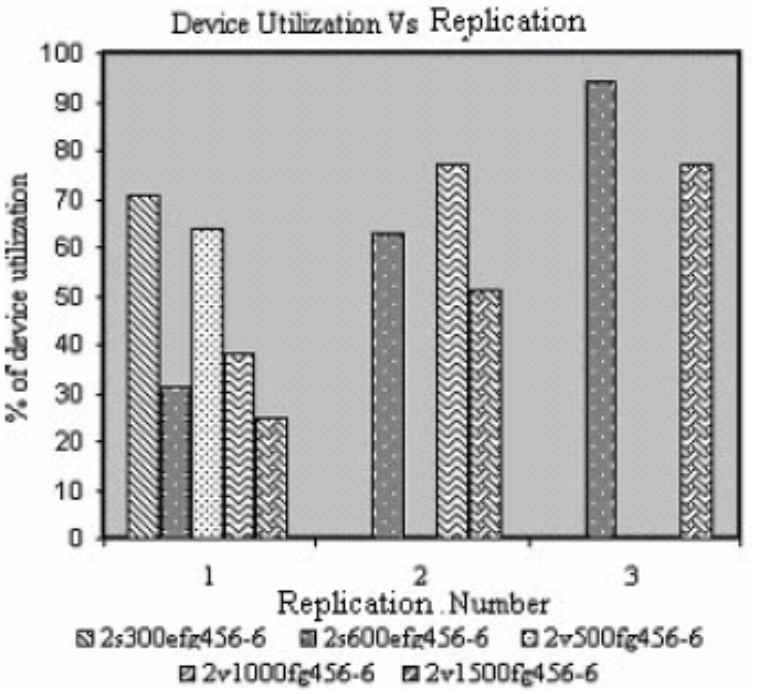
Table 4 indicates the effect of the replications on the speedup. While Table 1 indicated that as the area occupied by the hardware increases by as many times as the replication factor, the need of resources also increased due to replication. To have an overall gain the speedup must also increase substantially. Table 4 shows that the speedup also increases by almost as many times as the replication factor. The speedup and area occupied have a linear relationship. So we can achieve the speedup that is better than the any known implementations.

The throughput and the clock frequency of different architectures reported in the literature for implementing Blowfish algorithm are compared in Table 3. The throughput for the DRIL architecture is found to perform efficiently at higher speed and much higher throughput. The comparison has been with for DRIL architecture with a replication factor of one. We can observe that DRIL compares well with the ASIC architectures [8, 9, 10] and has a higher throughput than other FPGA architectures reported in [12].

6 Conclusions

A new efficient dynamically reconfigurable-replicated flexible design, DRIL architecture for blowfish algorithm is proposed in this paper. The features of the blowfish algorithm is exploited to produce a reconfigurable and pipelined design.. Besides hardware, the DRIL architecture involves a software design, as well. The software design controls the operations of the DRIL hardware and interacts with the applications that employ this architecture for encryption. It also ensures the dynamically reconfigure key generation unit into encryption unit. The replication technique used in the design helps to increase the throughput and brings about the maximum utilization of the device. Application specific configurable hardware can be made to coexist with the DRIL hardware on the same device owing to the replication feature. DRIL is validated, analyzed and the results prove it to be flexible, efficient in hardware utilization and have better performance with high encryption speed and

throughput. DRIL architecture could be replicated to three times on the SPARTAN 2E and VIRTEX II devices. For this replication factor, the architecture had a maximum frequency of 73.828 MHz with throughput of 778.845 Mbps on the SPARTAN 2E and a frequency of 146.515 MHz with throughput of 1545.654 Mbps on the VIRTEX II FPGA device. Although the reconfiguration time can be a major concern, as it is done only once for key generation, the encryption will be much faster. As the technology of FPGAs is yielding better devices, the authors believe that this may not be a major concern in the days to come.



References

1. William Stallings: Cryptography and Network Security principles and practice, 3rd ed., Pearson Education, (2003).
2. Bruce Schneier,: Applied Cryptography: Protocols, Algorithms and Source code in C, 2nd ed., John Wiley & Sons, (1996).
3. Bruce Schneier,: Description of a New Variable-Length Key, 64 bit Block Cipher (Blowfish), Proc. of Fast Software Encryption Cambridge Security Workshop, Springer-Verlag, (1994), 191-204.
4. Bruce Schneier: The Blowfish Encryption Algorithm -One year later, Dr. Dobb'S Journal, Sept. (1995).
5. K. Compton and S. Hauck.: Reconfigurable Computing: A Survey of Systems and Software, ACM Computing Surveys, vol. 34, no. 2, June (2002).
6. T. Wollinger and C. Paar: How Secure Are FPGAs in cryptographic Applications?, Int'l Conference on Field Programmable Logic and Applications (FPL 2003), Lisbon, Portugal, Sep. 1-3, (2003).

7. Y. K. Lai and Y. C. Shu: A novel VLSI architecture for a variable-length key, 64-bit blowfish Block cipher, Proc. of IEEE Int'l Workshop on Signal Processing System Design and Implementation (SiPS 99), Taipei, Taiwan, (1999), .568-577.
8. Y. K. Lai and Y. C. Shu: VLSI Architecture Design and Implementation for Blowfish BlockCipher with Secure Modes of Operation, Proc .of IEEE Int'l. Symposium on Circuits and Systems (ISCAS 01), May (2001), 57-60.
9. Michael C J Lin and Youn-Long Lin: A VLSI Implementation of the BlowfishEncryption/Decryption Algorithm, Asia and South Pacific Design Automation Conference (ASPDAC),Yokohama, Japan, Jan. (2000).
10. S.L.C Salomao, J.M.S.de Alcantra, V.C.Alves and A.C.C Vieria: SCOB, a soft-core for the Blowfish Cryptographic algorithm, Proc. IEEE Int'l Conf. Integrated Circuit and System Design, (1999), 220-223.
11. H. Singpiel, H. Simmler, A. Kugel, R. Manner, A.C.C. Vieira, F. Galvez-Durand, J.M.S. de Alcantara, V.C. Alves: Implementation of Cryptographic Application on the Reconfigurable FPGA Coprocessor, 13th Symposium on Integrated Circuits and Systems Design (SBCCI'00) , Manaus, Brazil, September 18 - 24, (2000)
12. Pawel Chodowiec, Po Khuon and Kris Gaj: Fast implementations of secret-key block ciphers using mixed inner- and outer-round pipelining, Proc. of the 2001 ACM/SIGDA 9th Int'l symposium on Field programmable gate arrays, Monterey, California, Feb. 11-13, (2001), 94-102.
13. A J. Elbirt, W Yip, B. Chetwynd and C. Paar: An FPGA implementation and performance evaluation of the AES block cipher candidate algorithm finalists, Proc. of the 3rd Advanced Encryption Standard (AES) Candidate Conference, New York, Apr. 13-14, (2000).
14. R. R. Taylor and S. C. Goldstein: A high Performance flexible architecture for cryptography, First Int'l Workshop on Cryptographic Hardware and Embedded Systems (CHES'99), Springer-Verlag Lecture Notes in Computer Science, vol.1717, Aug. (1999), 231-245.
15. J. Kaps and C. Paar: Fast DES implementations for FPGAs and its Applications to a Universal Key Search Machine, 5th Annual workshop on Selected Areas in Cryptography (SAC'98), Springer-Verlag, Aug. (1998)
16. Cameron Patterson: High Performance DES Encryption in Virtex(tm) FPGAs Using Jbits(tm), Proc. of the 2000 IEEE Symposium on Field-Programmable Custom Computing Machines, Napa Valley, California, April 17-19, (2000).
17. A. Hodjat and I. Verbauwhede: A 21.54 Gbits/s fully pipelined AES processor on FPGA, IEEE Symposium on Field-Programmable Custom Computing Machines, April (2004).
18. V.Fischer and M.Drutarovsky: Two Methods of Rijndael implementation in Reconfigurable hardware, 3rd Int'l Workshop on Cryptographic Hardware and Embedded Systems (CHES 01), Springer-Verlag Lecture Notes on Computer Science, May (2001), 77-92.
19. Backup for Workgroups, Automatic backup and restore software, http://www.backup-forworkgroups.com/A.
20. Encryption Plus Hard disk, http://www.pcguardiantechnologies.com/Encryption_Plus_Hard_Disk/
21. Encryption Tools, http://www.parisien.org/pet.htm.
22. Ultra Scan, True Identity biometrics, http://www.ultra-scan.com.
23. Cryptography: Protecting your Geographic Data http://www.geodyssey.com/cryptography/ cryptography.html.

Appendix

Table 1. Device Utilization

DEVICE	D	SLICES	SLICE FLIP FLOPS	LUTS	IOBS	FIT
	SPARTAN 2E					
2s300efg456-6	1	71	26	61	43	Yes
2s600efg456-6	1	31	11	27	43	Yes
2s300efg456-6	2	143	53	121	43	No
2s600efg456-6	2	63	23	54	43	Yes
2s300efg456-6	3	214	80	183	44	No
2s600efg456-6	3	94	35	80	44	Yes
	VIRTEX 2					
2v500fg456 -6	1	64	24	56	53	Yes
2v1000fg456 -6	1	38	14	33	43	Yes
2v1500fg456 -6	1	25	9	22	36	Yes
2v500fg456 -6	2	128	48	110	54	No
2v1000fg456 -6	2	77	29	66	44	Yes
2v1500fg456 -6	2	51	19	44	36	Yes
2v500fg456 -6	3	193	73	166	55	No
2v1000fg456 -6	3	115	43	100	45	No
2v1500fg456 -6	3	77	29	66	37	Yes

Table 2. Frequency table

DEVICE	D	MIN PERIOD (ns)	MAX FREQ (MHz)	MAX THROUGHPUT (mbps)	FIT
	SPARTAN 2E				
2s300efg456-6	1	13.545	73.828	259.615	Yes
2s600efg456-6	1	13.545	73.828	259.615	Yes
2s300efg456-6	2	13.545	73.828	519.23	No
2s600efg456-6	2	13.545	73.828	519.23	Yes
2s300efg456-6	3	13.545	73.828	778.845	No
2s600efg456-6	3	13.545	73.828	778.845	Yes
	VIRTEX 2				
2v500fg456 -6	1	6.825	146.515	515.218	Yes
2v1000fg456 -6	1	6.825	146.515	515.218	Yes
2v1500fg456 -6	1	6.825	146.515	515.218	Yes
2v500fg456 -6	2	6.825	146.515	1030.436	No
2v1000fg456 -6	2	6.825	146.515	1030.436	Yes
2v1500fg456 -6	2	6.825	146.515	1030.436	Yes
2v500fg456 -6	3	6.825	146.515	1545.654	No
2v1000fg456 -6	3	6.825	146.515	1545.654	No
2v1500fg456 -6	3	6.825	146.515	1545.654	Yes

Table 3. Comparison with the related work r is the replication factor

IMPLEMENTATION	TYPE	FREQ (MHz)	THROUGHPUT (Mbps)
Salomao et al [10]	VLSI	50	----
Lai et al [8]	VLSI	72	-----
Lin et al[9]	VLSI	66	266
Chodowiec et al [12]	FPGA	10	40
DRIL (R=1)	FPGA (SPARTAN2E)	73.828	259.615
DRIL (R=1)	FPGA (VIRTEX 2)	146.515	515.218

Table 4. Speedup follows replication scale

REPLICATION (D)	# NO OF CLOCK CYCLES REQUIRED FOR ENCRYPTION	SPEEDUP WITH RESPECT TO NO REPLICATION (R=1)
1	23579	1.00
2	11846	1.99
3	7939	2.97

Efficient Architectural Support for Secure Bus-Based Shared Memory Multiprocessor

Khaled Z. Ibrahim

Department of Electrical Engineering,
Suez Canal University. Egypt
kibrahim@mailer.eun.eg

Abstract. Tamper-evident and tamper-resistant systems are vital to support applications such as digital right management and certified grid computing. Recently proposed schemes, such as XOM and AEGIS, assume trusting processor state only to build secure systems. Secure execution for shared memory multiprocessor is a challenging problem as multiple devices need to be trusted.

In this work, we propose a framework for providing secure execution on a bus-based multiprocessor system that tackles the key distribution problem, the overhead of encryption/decryption and the memory integrity overheads. We show how to remove the encryption/decryption latencies from the critical path of execution using *pseudo* one-time-pad.

While verifying the integrity of all memory transactions, we use a special buffer to check for replay on a random set of memory lines. Replay can be detected with certainty of 99.99%, even if the lines replayed are less than 1%.

1 Introduction

Secure execution has grasped the attention of both academia and industry [1,2,3]. Applications of secure computing include digital rights protection, certified execution, and copy-proof software.

Secure Execution encompasses confidentiality and integrity. Confidentiality maintains secrecy of the data and code during execution, while integrity prevents malicious alteration of data or code. To violate security, an adversary may try to alter memory, inject/block transactions on the system buses or alter processor state.

Software solutions for secure execution faced many failures because software can be easily analyzed to detect its secrets. Newly proposed secure processors protect secrets using tamper-proof hardware mechanisms, thus offering an appealing opportunity for robust secure computing.

Architectural support gives the chance of providing security while maintaining good performance. A notable such architectures are XOM [2] and AEGIS [3]. Both assume that only the processor core has the trusted state. The OS and other parts of the system are assumed insecure. These schemes guarantee the privacy and the integrity of the software.

T. Srikanthan et al. (Eds.): ACSAC 2005, LNCS 3740, pp. 640–654, 2005.

The need for higher performance drives supporting secure computing in multiprocessor machines. Trusting multiple processors and devices complicates designing secure environment due to the need for key distribution mechanism.

This proposal assumes that a group of devices, determined by the software vendor, are trusted to execute the software. These devices do not need to have a built-in shared secret key. Instead, a shared secret key is distributed and is used later as a session key.

In the context of multiprocessor system, this proposal targets an efficient solution to the following issues:

- Protecting the privacy of software through encryption/decryption, thus the software cannot be run by a device that is not authorized to execute it. Our scheme always removes encryption/decryption latency from the critical path of execution using a pseudo One-Time-Pad (OTP). We also introduce an efficient key distribution mechanism.
- Guaranteeing integrity of the distrusted memory system. While verifying the integrity of all transactions with the distrusted memory, our proposal checks for replay attacks only for random set of transactions. We show that with minimum overhead the chance of not detecting replay attacks is negligible.

The remaining of this paper is organized as follows: Section 2 introduces our proposed software privacy scheme and also the key distribution mechanism for a multiprocessor system; Section 3 presents the proposed integrity verification of memory; Section 4 explores extending instruction set to support secure computing; related work is presented in Section 5. We conclude in Section 6.

2 Protection of Software Privacy

Protecting the privacy of software is done through encrypting software such that only the processor authorized to use this software can decrypt it. Data is stored in the memory in an encrypted form. Encryption/decryption is usually done using a secret key, for instance using AES [4]. Decrypting data using a secret key can expose additional latency in the critical path of cache misses. Communicating the key with the processor is done through asymmetric key cryptography, for instance using RSA [4]. The asymmetric key encryption is much more expensive than for symmetric key.

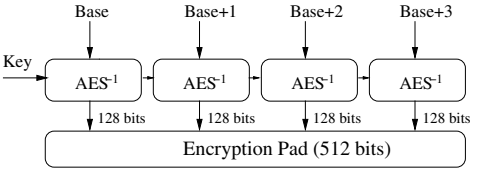


Fig. 1. One-Time-Pad generation

One-Time-Pad (OTP) has been shown successful to move the latency of decryption from the critical path of a cache miss [5,3,6]. Figure 1 shows the technique of producing the pad used in OTP scheme. To encrypt a message m using a pad p, the pad is xor-ed with the message. Decryption is done through xor-ing the encrypted message with the same pad.

The *base* for the pad can be chosen to be sequence numbers [7], or a combination of addresses and sequence numbers [6,5]. The initial vector for encryption is the secret key. Depending on address alone has the problem of generating the same pad every time for the same address. Common sequences of stored values can expose the encryption pad [6]. Despite this disadvantage, this scheme has the advantage of starting the pad computation as early as knowing about the miss, thus allowing a complete overlap of the cache miss and the pad computation. Using sequence number as an OTP base makes the pad not repeated for the same cache line, thus making it difficult to detect common and repetitive sequences. The disadvantage is that the sequence number to generate the pad may not be cached within the processor chip and so the pad computation will start after receiving the sequence number from the memory. The pad computation will not be overlapped with the whole miss time, and thus exposing most of the decryption latency.

Caching sequence numbers cannot scale if the data set is large or in a multiprocessor setting. Shi [7] showed that shared memory applications can slow down significantly (55% on average) even with OTP.

We use addresses in generating OTP because we want to prevent malicious move of the memory content. Unlike virtual addresses, using physical addresses restrict the OS in the relocation of memory physical frames. Accordingly, virtual addresses are preferred in OTP pad generation.

Shared address space can be created among different processes using either virtual space aliasing or fixing virtual to physical addresses among processes. In the first technique, each process has different image of virtual shared addresses which are mapped to the same physical addresses. In the later technique, all processes sharing a certain arena of addresses have the same mapping of virtual to physical mapping. This technique is easier to program and is widely supported in modern OS. In both techniques, the OS should be able to move physical pages out of memory and to relocate virtual pages and to update the mapping for all processes. In the proceeding discussion, we assume that there is no virtual memory aliasing for shared memory, i.e. two virtual pages cannot be mapped to the same physical frame. This allows using virtual addresses as the base of OTP generator. For non-shared data, using the same virtual address by different processes, as a base for the pads, does not cause problem because this virtual address will be mapped by each process to a distinct physical locations in the memory. The pads are also dependent on the initial vector (the secret key).

An orthogonal but related problem is the information leakage on the address bus [8]. The control flow information can be exposed through monitoring the addresses generated on the bus. Zhuang et al. [8] proposed permuting addresses during execution.

In the following section, we introduce our proposed model to protect the privacy of both data and address information.

2.1 Securing Data and Address Information

In multiprocessor environment, providing secure environment requires trusting multiple processors [9,7]. The system buses are vulnerable to security attacks because they are physically accessible. For instance, the addresses on the bus can reveal control flow information and thus code signatures.

Interestingly, solving the security problem for data and addresses can be much related. OTP schemes requires not using addresses alone because the pad although different (one-time) for different memory lines, it is the same for the same memory line. If we hide that the accesses are to the same memory line, along with having different pads for different memory line and then we will generate a *pseudo-OTP* that makes cryptanalysis difficult.

The layout of our proposed solution is as follows

- To encrypt data the base for OTP will be based solely on virtual address. The key used for encryption is shared among devices (processors) authorized to access these memory pages. Virtual addresses are fortunately known at the beginning of a cache miss.
- The physical addresses to request memory line will be protected by encrypting them using another one-time-pad engine. The base of this OTP-based address encryption is a sequence number that is derived by the count of completed bus transactions. The secret key used to encrypt addresses is different from the key used to encrypt data and is shared among processors and memories.

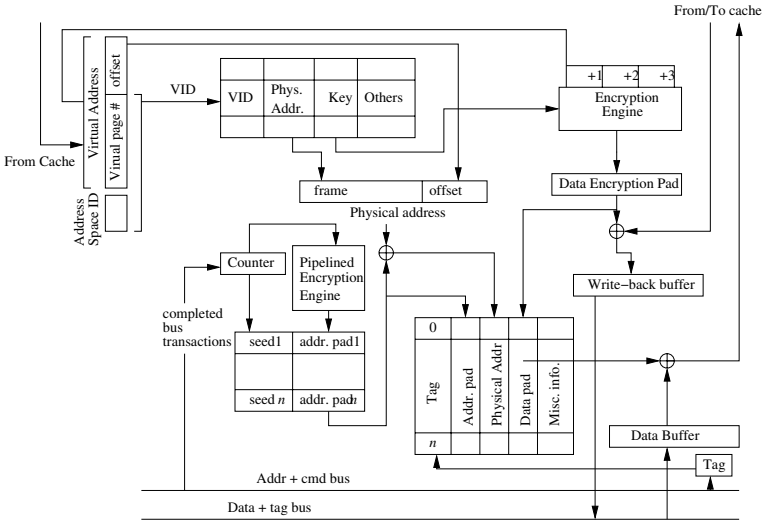


Fig. 2. Microprocessor encryption engine

Although we use the same pad for the same memory line, this is not exposed because the same line appears with different physical addresses due to encryption. Additionally, encrypting addresses eliminates information leakage, such as control flow and software signatures.

The sequence numbers can be generated independently by all the processors sharing the system buses and derived by the number of completed transactions. These transactions are similarly seen by all processor on a snoop-based cache coherent system. Each sequence number is kept for the life time of its corresponding memory transaction, and is never needed after that transaction is completed. Caching for the pads and the sequence numbers are kept in the MSHR while the miss is handled.

The address pads are taken from the critical path of execution by generating pads for future sequence numbers and then queuing of these pads. This will be especially easy with a pipelined implementation of the encryption engine. These pads are not only used for satisfying a processor's own misses but also for snooping to the bus. Although we cannot guarantee that encrypted addresses will not have collision, this will not be a problem as the response is usually accompanied by the operation tag (unique identifier) and not the physical address that may be repeated. We can still support a mixture of encrypted and non-encrypted addresses. This requires differentiating between these transactions on the control lines before the address is exposed on the bus.

It should be noted that encrypting addresses do not alleviate the need of distrusting the memory state. It merely creates a secure communication channel with the memory. The contents of the memory still need to be checked for integrity and encrypted for confidentiality.

Figure 2 shows the modifications needed for the processor core, while Figure 3 shows the engine from the memory side. The memory system can be implemented based on *intelligent memory* [10] that embeds a simple processor

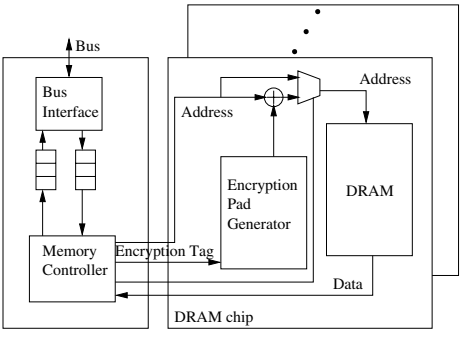


Fig. 3. Memory with embedded address-encryption engine

in the memory side. Typically a system has multiple memory modules; all of them should support encrypted addresses.

The following section introduces the mechanism for sharing the secret keys for encrypting/decrypting addresses and data.

2.2 Secure Key Distribution and Management

Symmetric key cryptography is usually used to encrypt and decrypt the application code and data because of its performance. In uniprocessor system, the processor should have a pair of asymmetric keys, a public key and a private key. Exchanging the symmetric key used for encrypting the application software is done through encrypting the symmetric key using the processor public key. Only the processor can decrypt the symmetric key using its private key.

For secure multiprocessor systems, the above technique is extended by encrypting the symmetric key by all the public keys for the devices authorized to access the application software [9], thus generating multiple encrypted version for the symmetric key.

In this section, we introduce a mechanism for sharing a symmetric key that comprise having a single key (possibly requiring a smaller space), securing the group membership of devices allowed to access the software, and finally being computationally feasible.

Let a system of n devices need to share a secret key k. Each device i defines a pair of asymmetric pair of keys; e_i: a public key, d_i: a private key. These keys have the property that given e_i it is computationally infeasible to determine the corresponding d_i.

Let $a_i = E_{e_i}(k)$, $1 \leqslant i \leqslant n$ be the secret key encrypted by the public key of each member in the group of devices that need to share the key k. Let each device i publicly define an integer p_i such that $\gcd(p_i, p_j) = 1$; for all $j \neq i$; *i.e.*, p_1,$p_2, \cdots, p_n$ are pairwise relatively prime. It is required to find the solution x of the following simultaneous congruences:

$$x \equiv a_i \pmod{p_i}, 1 \leqslant i \leqslant n \tag{1}$$

Finding the solution x given the set of a_i and p_i, $1 \leqslant i \leqslant n$, satisfying the simultaneous congruences is called Chinese Remainder Theorem (CRT) [11]. The solution has a unique solution modulo $p = \Pi_{i=1}^{k} p_i$. The solution x to the simultaneous congruences 1 is in the range $[1, p-1]$ and may be computed as:

$$x = \left(\sum_{i=1}^{n} (p/p_i) * a_i * m_i \right) \bmod p \tag{2}$$

where $m_i = (p/p_i)^{-1} \bmod p_i$

An efficient way of computing the solution x is through Garner's Algorithm [4].

The software vendor has to do the following procedure: a) The vendor determines the set of devices that are authorized to decrypt/use the software; b) The

vendor picks secret key k and encrypts the software with this secret key. This key is better varied for the same software that is distributed to multiple hardware groups; c) Based on the CRT, given the available public keys and prime numbers, the common key x is computed and associated with the software.

The system using the software does the following procedure: a) Each device i authorized to use the software takes the common key x to reach its encrypted version of the shared secret, $a_i = x \bmod p_i$; b) Each device i decrypts a_i to find the session key, k.

Two keys are needed; one key for encrypting and decrypting the software and the application data, and the other key for encrypting and decrypting addresses. While the first key is distributed with the software, the second key is per system, and may be changed only when the system components are changed. Key extraction involves a public key decryption, which can take thousands of cycles. It should therefore be done infrequently. Using a key per memory page can be tolerable, provided there are very large pages and many TLB entries.

3 Integrity of the Memory

Integrity of memory targets the detection of modified/corrupted memory state. The adversary may try to modify memory contents in an attempt to gain information about the running software or to modify its behavior. As the adversary may be able to modify the memory state, the processor authorized to access the memory state should be able to detect it so as to stop the execution. The authenticity of memory state can be based on message authentication code (MAC) as proposed by Lie et al. [2]. MAC techniques usually hash the data and the address and then encrypt the hash using a secret key. The main problem with using MAC is that valid memory state can be *replayed*; as MAC does not guarantee that the memory state is the most recent.

The integrity information should hash address along with data of the memory to protect against malicious move of memory. Using virtual addresses allows the OS to move memory pages freely without any need to change the integrity information associated with them. On the other hand, virtual addresses cannot be used to snoop on bus transactions. Physical addresses are used to snoop to the transactions on the bus, but the integrity information will need to be changed every time the OS changes the virtual to physical mapping.

To solve the integrity problem as well as preventing replay of authentic data, a trusted state of the memory is saved inside the processor. Whenever data is loaded into the processor, data integrity is checked against the trusted state. To minimize the time complexity of verifying the integrity of memory, tree-based structures are proposed [6,5,3,7]. Even these tree structures have expensive space requirements and may not be entirely cached within the processor. Integrity verification can be computationally expensive if the verification data are not entirely cached.

Another approach, proposed by Suh et al. [5,3], is to log transactions and then to verify the integrity of that set of transactions. This reduces the runtime

overhead. Tampering can be detected only at checkpoints. Checking the integrity requires loading all memory blocks which is a very expensive operation. Another problem with this approach is that part of the memory may be unavailable (for instance, deallocated) at verification time.

Applying these approaches to multiprocessor [7,12] setting requires distributing the integrity checking from one processor to multiple cooperating processors. A problem with this approach is that it requires integrity checking using physical addresses because they are the only information shared on the bus.

3.1 Proposed Memory Integrity Model

We propose guaranteeing anti-replay on opportunistic basis. By "opportunistic," we mean that we fully verify only those transactions for which we have complete integrity information within the processor. For memory lines which we do not have information, we verify only the data integrity, without making sure that these lines are replayed. Memory lines are always stored in the memory with integrity information (MAC and version number).

Details of our proposal go as follows:

- Each processor keeps track of the integrity information for a random set of memory lines produced (written) by this processor in an integrity buffer.
- If the line brought to cache hits in the buffer that holds integrity information, integrity is checked using the cached state. If not cached, only MAC is used for verification.
- The integrity verification is carried out for transactions on the bus if they hit in the integrity buffer, even if not requested by the same processor.
- An integrity exception does not need to be precise, *i.e.*, we do not need to tie the integrity exception with a certain instruction. In fact, the integrity exception is better to be imprecise. This hides transaction that is caught unauthentic, especially that we do not verify all transactions. Additionally, we authenticate transactions for other processors, and so we cannot provide precise exceptions. We still need to be careful about tying the exception with the process that caused it.

A bus is considered authentic if all transactions on the bus are seen by all processors. Authentic bus is assumed by cache coherence protocols. In Section 2.1, independent counters for the number of completed bus transaction are used as the base to generate address pads. These pads are used to decipher the address. If the numbers of completed transactions are different, then pads will not be synchronized and addresses will be deciphered differently.

In this sense, the scheme described in Section 2.1 provides partially authentic bus. We do not have a fully authentic bus because we do not guarantee that the same transactions are seen by all the devices (not only the count of transactions). Correct execution is not guaranteed in a cache-coherent protocol if not all the transactions are seen by all devices.

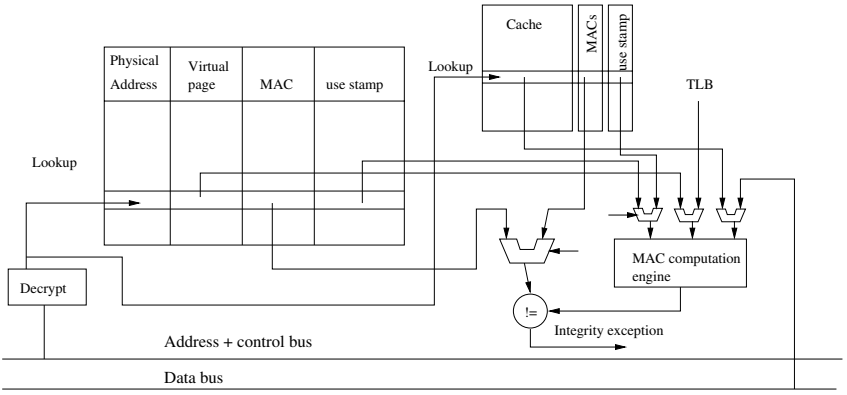


Fig. 4. Integrity checking in a snooping-based bus system

3.2 Integrity/Anti-replay Snooping Buffer

The integrity buffer is shown in Figure 4. The buffer contains the following entries: Physical address, Virtual page number, Use-stamp and Message Authentication Code (MAC). Use-stamp number is incremented each time a modified copy of cache line is sent to memory. MAC is the encrypted hash of the cache line, the use-stamp, and the virtual address.

Replacement policy in the integrity buffer goes as follows:

- A page invalidated from the TLB should invalidate the corresponding entries in the authentication cache. Accordingly, the TLB replacement is better to be random.
- Cache lines written back, due to replacement for instance, are inserted in the integrity buffer. Lines downgraded from dirty to shared due to a read by another cache are also inserted into the buffer.
- Replacements can be pseudo-random, or could favor the authentication of lines with high use-stamp values. Alternatively, the replacement could be based on the decay of use concept [13].
- A line exclusively requested on bus is verified, and then evicted from the integrity buffer. The new owner will be responsible for the integrity checking of this line. An adversary may want to force a line out of the integrity buffer by faking an exclusive request on the bus. Forcing eviction will be extremely difficult because the addresses are encrypted, and an encrypted version of the address cannot be reused later.
- The application can specify critical addresses to be kept track of during the execution. These addresses will not be replaced. The use of this facility should be minimal so as not to overflow the integrity buffer.

3.3 Transaction Integrity *vs.* System Integrity

In our work, we propose providing integrity with opportunistic anti-replay detection. The processor will check integrity and limit anti-replay detection to a random set of memory lines produced by this processor. The application may also decide certain addresses to be provided continuous full integrity.

To assess the effectiveness of the proposed scheme: Let the probability of detecting a memory line replay be p_a. Replay is checked for lines which are present in either the cache or the integrity buffer. The count of lines that can be checked for replay is small if compared with the application dataset.

The probability p_a is dependent not only on ratio between the size of cached data to the total data set, but also on the spatial locality of references. The chance of not detecting a replay violation for a single memory transaction is $(1 - p_a)$.

If the number of transactions to be replayed is α, then the chance of not detecting replay violations is $(1 - p_a)^\alpha$. If α is small then, there is a good chance that replay will not be detected; thus we add the support of specifying a small set of addresses that are checked for integrity all the time and not replaced from the integrity buffer. If α is large, *i.e.*, large space is replayed, then there is a good chance that replay will be detected because $(1 - p_a)^\alpha$ will tend to be zero. The opportunistic replay check will be enough to provide a high level of confidence that replays are detected.

Replay attack will be especially difficult with our scheme because of the following reasons:

- Replay attacks are a concern only for data that are modifiable by the program. These data are kept for replay check in the integrity buffer described earlier for a random period of time. It is difficult for the adversary to tell when replay will not be detected.
- The producer-consumer and migratory sharing relations cannot be detected on the address-bus because addresses are encrypted.

3.4 Evaluating Integrity/Anti-replay Coverage

In this section, we study the effectiveness of the proposed integrity scheme. As discussed earlier, our scheme verifies the integrity of all transactions but verifies replay for part of transactions.

Table 1. Benchmarks and their problem sizes

bench	size	bench	size
LU	512×512	Ocean	258×258
FFT	262141	Water-nsquared	512 mols
Cholesky	tk15.O	Water-spatial	512 mols
Radix	1M integers	Volrend	head
Barnes	16K particles	Raytrace	car

The simulator used is based on SimOS [14]. We used MIPSY processor model with L2 cache size of 256K bytes. The system is a quad-processor bus-based system with invalidation-based cache coherence protocol.

The benchmarks are taken from SPLASH2 suite [15]. Table 1 lists the used benchmarks and their problem sizes.

We studied the percentage of transactions that are verified for integrity and anti-replay against all transactions that carried data on the bus. We excluded writeback transactions because they can be trivially verified while being transmitted to the memory. Upgrades are not counted as they do not carry data. The transactions on the bus are verified either by the L2 cache having the same data or by the integrity buffer discussed in Section 3.2. We have chosen two integrity buffer sizes: the first with 16K entries, the second with 128 entries. Both are 16-way set associative.

Figure 5 shows the decomposition of the transactions verified on the bus. The average transactions verified by the caches exclusively are 8% for the 16K entries and 12% for the 128 entries. The average transactions verified by both the caches and the integrity buffers are 10% and 5% for the buffer sizes of 16K and 128 , respectively. The transactions verified by the integrity buffer are further decomposed into; (a) transactions for data extracted from one owner cache to be read by another cache and (b) transactions verified after the line is evicted from the cache. The first component is independent of the integrity buffer size and contributes to the verification on average by 34%. The second component rose from an average of 7% for 128 entries to 23% for 16K entries.

The average transactions verified totaled 75% for 16K buffer. We still check the integrity of the remaining transactions, but we cannot check replay. It is notable that some of the remaining transactions are for read-only data, such as

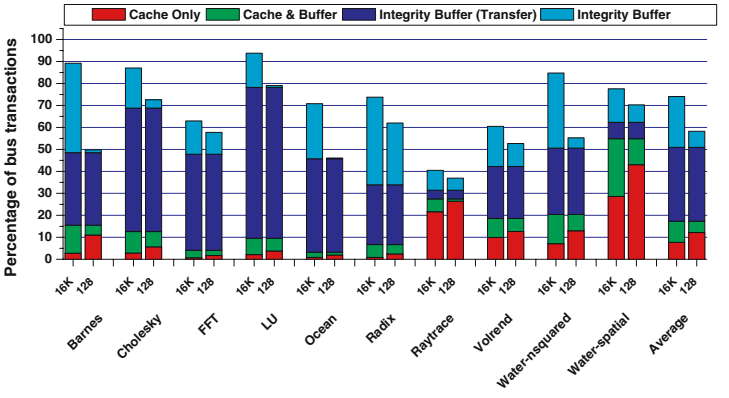


Fig. 5. Integrity/Anti-replay verification of bus transaction excluding writebacks and upgrades for two buffer sizes 128 and 16K

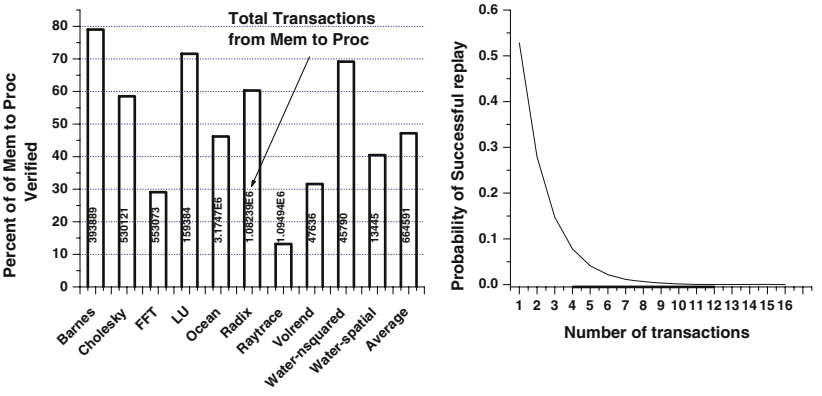


Fig. 6. Checking anti-replay considering transactions from memory to processor. a) Percentage of transaction. b) Probability of replay success

text segments. These data do not have multiple versions so replay verification is not needed.

Figure 5 gives insights to why sequence number can cause problem if used in generating OTP to encrypt data. Large portion of transactions are cache-to-cache, *i.e.*, caching sequence numbers will usually have a stale data. This shows the importance of not basing OTP on sequence number. As discussed earlier, our proposal encrypts address and bases OTP on virtual addresses alone.

Figure 6 considers 16K-entry integrity buffer, and only transactions coming from memory (*vs.* cache-to-cache). Figure 6.a shows the percentage of transactions fully verified, as well as the count of these transactions. Figure 6.b shows the probability of success for replaying memory transactions *vs.* the number of transactions. As shown in figure, replaying large amount of memory has a negligible chance of success. Actually, replaying more than sixteen lines has less than 0.01% chance of not being detected. As discussed earlier, the transactions verified are not known by the attacker. Sixteen transactions are negligible compared with the count of transactions, as shown in Figure 6.a. Replaying critical data (usually small amount) can be protected against by explicitly forcing entries in the buffer, as discussed earlier.

4 Instruction Set Architecture (ISA) Support

In this section, we will present enhancements to the ISA that enable software to control secure execution.

Broadcast Address Key (bak) instruction is used to access the memory location where the group key for addresses is stored. This instruction should be privileged instruction that is run only by the OS. Each device starts computing

the secret session key for addresses. The counters for generating address pads are also reset by this instruction. This key needs to be changed when new component need to be added to the system, for instance new DRAM module.

Broadcast Data Key (*bdk*) instruction is used to access the memory location where the group key for encrypting/decrypting data is stored. This instruction is not a privileged instruction. The processor can start secure execution after reaching a special instruction *Start Secure Execution (sse)*. The secure execution could be terminated upon executing an instruction *End Secure Execution (ese)*.

The use of the computed keys starts after *sse* instruction is executed. The instruction *sse* works as a hardware barrier that guarantees that session keys are computed. If *bak* or *bdk* instruction were not executed, then the *sse* instruction should cause exception.

For authentication buffer two instructions are defined *Monitor Integrity* (*mi*) and *Relinquish Integrity Monitoring* (*rim*). These two instructions can be used by the application to enforce adding or removing lines to/from the integrity buffer. Lines added to the integrity buffer are continuously monitored and guaranteed not to be replayed.

Verify Authority (*va*) instruction is used by the processor to check if it has access rights on a certain memory region. This instruction specifies a memory address that has the software/memory-page data secret key encrypted by the same key. The processor loads the memory encrypted key and then decrypts the key. If the key found matches the secret key, available in the TLB, the execution resumes normally; otherwise, an exception is generated and execution is aborted.

Complementary to these ISA enhancements, AEGIS [3] supports exchanging data with the external distrusted world. It additionally discusses how to implement secure context management.

It is should be evident that architectural security enhancements does not alleviate the need to security in the software layer. For instance, a database program needs to set properly the security privilege for different users, or the system can be easily compromised even on a secure hardware. Secure architectural support complements software security and does not replace it.

5 Related work

XOM model, proposed by Lie et al. [2], describes the architectural support needed for secure processor system. Numerous research papers [12,5,3,6,8,9] enhanced the basic model and evaluated its effect on performance. The problem of providing privacy and integrity for shared memory multiprocessor has been studied by Shi [7] and Zhang [9]. Unlike our scheme, Shi assumes the existence of secure OS kernel and build the integrity verification and encryption on physical address. Zhang [9] tries to remove the integrity check from the critical path through using Cipher Block Chaining mode of the Advanced Encryption Standard (CBC-AES). Clarke [12] extended schemes based on hash trees to verify the integrity of symmetric multiprocessor system. Integrity verification requires verifying a path from the memory line to the root of the whole memory

system. This tree structure is expensive both in space and time. In contrast, our proposed scheme verifies the integrity of all transactions and opportunistically verifies some memory lines for anti-replay.

The problem of information leakage on address bus is studied by Zhuang et al. [8]. Control flow and software signature information can be protected by permuting memory address space [8]. In our proposal, we encrypt the physical addresses to protect the privacy of the control flow and access pattern.

6 Conclusions

In this work, we address the problem of providing secure computation for bus-based shared memory systems.

Our scheme encrypts addresses in order to protect against the information leakage on address bus and also to make data encryption solely based on virtual address. We propose a mechanism called pseudo-OTP that enables steadily overlapping memory transactions with encryption/decryption. While maintaining distrusting the memory state, we secure a communication path to memory. The problem of distributing secret key among trusted processes is tackled based on Chinese Remainder Theorem.

The memory integrity problem is handled using simple MAC scheme with integrity buffer to verify a mixture of predetermined and random set of memory lines for replay. We introduced a mechanism of opportunistic anti-replay checking. We show that replay success chance tends to zero even if the count of replayed memory transactions are small.

References

1. : The Trusted Computing Platform Alliance. (http://www.trustedpc.com)
2. Lie, D., Thekkath, C., Mitchell, M., Lincoln, P., Boneh, D., Mitchell, J., Horowitz, M.: Architecural Support for Copy and Tamper Resistant Software. 9th Int'l Conf. on Arch. Support for Prog. Lang. and Operating Systems (2000) 168–177
3. Suh, G.E., Clarke, D., Gassend, B., van Dijk, M., Devadas, S.: AEGIS: Architecture for Tamper-Evident and Tamper-Resistant Processing. 17th Int'l Conf. on Supercomputing (2003) 160–171
4. Menezes, A.J., van Oorschot, P.C., Vanstone, S.A.: Handbook of Applied Cryptography. Volume 5th printing. CRC Press (2001)
5. Suh, G.E., Clarke, D., Gassend, B., van Dijk, M., Devadas, S.: Efficient Memory Integrity Verification and Encryption for Secure Processors. 36th Int'l Symp. in Microarchitecture (2003) 339–350
6. Yang, J., Zhang, Y., Gao, L.: Fast Secure Processor for Inhibiting Software Piracy and Tampering. 36th Int'l Symp. in Microarchitecture (2003) 351–360
7. Shi, W., Lee, H.H.S., Ghosh, M., Lu, C., Zhang, T.: Architecture Support for High Speed Protection of Memory Integrity and Confidentiality in Symmetric Multiprocessor. 13th Int'l Conf. on Parallel Arch. and Compilation Tech. (2004) 123–134
8. Zhuang, X., Pande, T.Z.S.: HIDE: An Infrastucture for Efficiently Protecting Information Leakage on the Address Bus. 11th Int'l Conf. on Arch. Support for Prog. Lang. and Operating Systems (2004) 72–84

9. Zhang, Y., Gao, L., Yang, J., Zhang, X., Gupta, R.: SENSS: Security Enhancement to Symmetric Shared Memory Multiprocessors. 11th Int'l Conf. on High-Performance Computer Architecture (2005) 352–362
10. Solihin, Y., Lee, J., Torrellas, J.: Prefetching in an Intelligent Memory Architecture using Helper Threads. 5th Workshop on Multithreaded Execution, Architecture, and Compilation (2001)
11. Chiou, G.H., Chen, W.T.: Secure Broadcasting Using Secure Lock. IEEE Trans. on Software Engineering **15** (1989) 929–934
12. Clarke, D., Suh, G.E., Gassend, B., van Dijk, M., Devadas, S.: Checking the Integrity of a Memory in a Snooping-Based Symmetric Multiprocessor (SMP) System. MIT LCS memo-470 (2004)
13. Kaxiras, S., Hu, Z., Martonosi, M.: Cache Decay: Exploiting Generational Behavior to Reduce Cache Leakage Power. 28th Int'l Symp. on Computer Architecture (2001) 240–251
14. Rosenblum, M., Bugnion, E., Devine, S., Herrod, S.A.: Using the SimOS Machine Simulator to Study Complex Computer Systems. Modeling and Computer Simulation **7** (1997) 78–103
15. Woo, S., Ohara, M., Torrie, E., Singh, J.P., Gupta, A.: The SPLASH-2 Programs: Characterization and Methodological Considerations. 22nd Int'l Symp. on Computer Architecture (1995) 24–36

Covert Channel Analysis of the Password-Capability System

Dan Mossop and Ronald Pose

School of Computer Science and Software Engineering,
Monash University, Clayton,
Victoria, Australia 3800
{dgm, rdp}@csse.monash.edu.au

Abstract. The Password-Capability System is a compact operating system with an access control mechanism based on password-capabilities. We show that the system is able to support several security paradigms which solve real-world problems not adequately addressed by conventional operating systems such as Windows and Unix. We show also that these paradigms are only effective if the system is free from covert channels. To this end, we carry out a covert channel analysis of the system and outline the elimination of all channels found.

1 Introduction

Conventional operating systems such as Unix, Linux and Windows offer support for basic security paradigms such as user logins, but fall short of supporting more advanced paradigms such as the principle of least authority and process confinement. This limits their ability to meet the needs of users operating in untrusted computing environments. The capability model [1] seems to offer scope for more flexible computer system security. The implementation of capability-based operating systems has traditionally taken the form of hardware tagging of capabilities, as for SWARD [2], or segregating of capabilities, as for SPEEDOS [3], MONADS [4] and Hydra [5]. This is required because one must guard against capability forgery, since the security of the system is dependent on them. However, these techniques do not support efficient mixing of data and capabilities without dedicated hardware, the benefits of which are described in [6].

Password-capabilities provide an alternative implementation of capabilities which can achieve this. A number of systems are based on the use of password-capabilities, including the Password-Capability System [7,8,9,10,11], Walnut [8], Mungi [12,13,14,15] and Opal [16]. This paper focuses on the Password-Capability System.

We show that the system supports both the principle of least authority and process confinement. We observe that the Password-Capability System's support for such security paradigms could be undermined by the existence of covert channels. To prevent this, we conduct a covert channel analysis of the system. We first present a formal model of the system based on that in [7]. We attempt to enumerate all possible communication channels in the formal model and show how to use this information to eliminate known covert channels from the paradigms.

T. Srikanthan et al. (Eds.): ACSAC 2005, LNCS 3740, pp. 655–668, 2005.

2 The Password-Capability System

In this section we describe and formalise the Password-Capability System. We detail the static state of the system and present the system calls which operate on it. We then describe the mechanism by which systems running the Password-Capability System can be connected across a network.

2.1 The Virtual Memory, Volumes and Objects

The Password-Capability System is an operating system with a global virtual address space, access to which is controlled by a mechanism utilizing password-capabilities. All entities such as data, files, processes, programs, in all such systems throughout the world, are considered to be objects. The virtual memory, M, is divided into volumes, V_v, each having a unique identifier, v. Each volume is typically a storage device. Each object $O_{(v,s)}$ resides on a volume, and is uniquely identified in the system by an object name (v, s). The object name is comprised of the identifier, v, of the volume and an object serial number, s, unique on that volume, assigned on the object's creation. Hence:

$$M = \{V_v : v \text{ is a volume identifier}\},$$
$$V_v = \{O_{(v,s)} : s \text{ is a serial number}\}.$$

New objects can be created on a volume. Each object when created has a single master capability, from which derivative capabilities can be created. The derivatives can never have more authority than their parents, and are destroyed if the parent is. Thus each object (v, s) has associated with it, a singly-rooted tree of capabilities, which we denote by $caps_{(v,s)}$. An object also has a data area, denoted $data_{(v,s)}$. We therefore write:

$$O_{(v,s)} = (data_{(v,s)}, caps_{(v,s)}).$$

2.2 Capabilities

Operations on objects are only permitted when a suitable password-capability is presented to the system. A capability (v, s, p) comprises an object name (v, s) and a randomly chosen password, p. Capabilities are simply values; therefore the security of an object depends on the infeasibility of guessing any capability for the object. This is ensured by choosing the password randomly and making it long enough that the probability of guessing one is negligible.

Each capability (v, s, p) permits some subset, $rights_{(v,s,p)}$, of all possible access rights to be exercised over the object. Aspects of objects that capabilities may give access to include the abilities to read or write a subset of the data in the object, the abilities to start, stop, and send messages to objects that are processes, the ability to derive new capabilities for the same object with a subset of the access rights. The capability also defines a window onto the data area of the object outside which the capability cannot be exercised. The window

is defined by a start offset, $start_{(v,s,p)}$, and a size, $size_{(v,s,p)}$. The start of the window is specified relative to that of the capability's parent, $parent_{(v,s,p)}$.

Apart from being a store of data and potentially a process, all objects also act as stores of money. This enables an integrated economic system to be created in which data and services can be traded, and resources managed in a familiar capitalist market model. Thus all objects are in effect bank accounts. The object's master capability indicates the object's total money in its *moneyword*. The moneyword for a capability, (v, s, p), is denoted by $money_{(v,s,p)}$. Derived capabilities' moneywords indicate withdrawal limits on the object's money. Thus the amount of money that can be withdrawn using a capability is the minimum of that of its moneyword, and that of all moneywords of its ancestors leading back to the master capability. The system periodically extracts a maintenance fee, proportional to the object's size, from the object. If this rent cannot be paid, the object will be deleted.

We can now give the formal specification of capability trees. A capability tree $caps_{(v,s)}$ for an object consists of a set of individual capability entries. We denote the entry for capability (v, s, p) by $cap_{(v,s,p)}$, so:

$$caps_{(v,s)} = \{cap_{(v,s,p)} : p \text{ is a password}\}.$$

A given capability entry, $cap_{(v,s,p)}$, is given by:

$$cap_{(v,s,p)} = (rights_{(v,s,p)}, start_{(v,s,p)}, size_{(v,s,p)}, parent_{(v,s,p)}, money_{(v,s,p)}).$$

2.3 Processes

Certain system calls allow objects to be used as processes. For instance, the *revive*() call, discussed further in Section 2.5, can be used to transform a new object into a process. Any process having a capability with the appropriate rights can use such calls to view an object as a process. When an object $O_{(v,s)}$ is used as a processs, the system enforces a particular format on its data area, $data_{(v,s)}$. The data area is split into a number of fields, including $type_{(v,s)}$, which stores information about the type of processor the process can run on, $status_{(v,s)}$ which indicates the status of the process (i.e. running, waiting, suspended, or terminated), and $pccap_{(v,s)}$ and $pc_{(v,s)}$ storing a capability to a code object and an offset into that object, respectively, which together form the process's program counter. Other fields in a process's data area are now described.

Processes can communicate with one another by passing short messages, or for larger messages, they can use an intermediate data object to which both have access. Message passing can be used to share capabilities to such an object. Messages sent to a process are stored in the $mailbox_{(v,s)}$ field of its data area. A process can choose to enter a waiting state, in which it will remain until a message is sent to it by some other process.

As part of the economic system, processes contain some cash for spending on immediate needs such as CPU time. Processes have a *cashword* field, denoted $cash_{(v,s)}$, in which their cash is stored. They can send this money to other processes, or store it in objects.

The Password-Capability System has a mechanism for addressing the confinement problem [17]. We defer description of this mechanism until later in the paper when it is treated fully. For now, it will suffice to observe that the mechanism requires each process to have an associated value which we call a *lockword*, and which we store in the $lock_{(v,s)}$ field of the process's data area.

For a process $O_{(v,s)}$, then, $data_{(v,s)}$ is given by:

$$data_{(v,s)} = (type_{(v,s)}, state_{(v,s)}, pccap_{(v,s)}, pc_{(v,s)}, \\ cash_{(v,s)}, lock_{(v,s)}, mailbox_{(v,s)}).$$

The Password-Capability System also provides a mechanism for abstract data type management.

2.4 System Calls

We now briefly describe the system calls which enable processes to interact with the virtual memory. With the exception of *make_obj*(), each of the system calls will only succeed if the calling process presents the system with a capability possessing the access right with the same name as the call. If the capability does not have the correct access right then the call will fail. The system does not require a capability to carry out the *make_obj*() call, hence any process can create objects. The reader is referred to [7] for a formal description of the calls.

The *make_obj*() call creates an object on a specified volume and gives it a serial number which is unique on that volume. It also creates a master capability for the object and returns it to the caller.

The *derive_cap*() call derives a new capability, the derivative, from an existing one. The existing one may be a master capability, or some other extant derivative. The derivative has a caller-specified subset of the access rights of its parent. Its window is a caller specified subset of its parent's. The caller can also specify a value for the derivative's moneyword. Capabilities can be deleted using the *delete_cap*() call. This deletes the capability and all of its derivatives. If the deleted capability is the master capability for some object then that object is destroyed as no further access to it is possible. The *rename_obj*() call can be used to create a new master capability for an object. In doing so, all other capabilities for the object are destroyed.

The *read*() and *write*() calls respectively read and write to the *data* field of an object. The operations can only be applied within the window of the presented capability. The *contract*() call decreases the size of this window. The *expand*() call increases the size of the object, rather than the window. However, any capability with the *end_of_object* right will have its window extended to the end of the object. The *cap_info*() call returns the value of a capability's *size* and *money* fields.

The status of processes can be controlled through several calls. *suspend*() will suspend a running or waiting process. *resume*() will cause a suspended process to resume running or waiting. *wait*() will cause a running process to wait for a message. *revive*() is used to transform a terminated process, or alternatively, a

new object, into a suspended process. *revive*() also enables the process's program counter to be set and cash to be passed to it by the caller.

send() can be used to send a short message, cash, or both to another process. These are placed in the recipient process's mailbox until *receive*() is called, at which point the cash is added to the recipient's mailbox and the message retrieved. As with all the system calls, *send*() is blocking; it will not return until the call has completed.

apply_lock() is used to modify the lockword of a process. This is used in the confinement mechanism discussed later in this paper. The *process_info*() call is used to discover a process's *status* and its cashword.

2.5 Networking

The Password-Capability System is designed to be global, with all systems running it sharing the same virtual memory. This raises the issue of how such systems are connected across a network. The mechanism is quite simple. The systems at each end of a connection implement a network buffer. A buffer appears as an ordinary object to processes on the same system. The system can distribute capabilities to this buffer object having, for instance, the *read* and *write* access rights. Any process given such a capability can read and write to the buffer. When a buffer is written to, the data will be forwarded to the corresponding buffer at the opposite end of the connection. The two systems can secure the connection using standard techniques. The mechanism is essentially transparent to processes in the system and does not feature in our analysis.

3 Security Paradigms

In this section we describe two important security paradigms supported by the Password-Capability System: the *principle of least authority*, and *process confinement*. We demonstrate through examples that they solve real-world problems not adequately addressed by conventional operating systems. Later sections discuss the effect of covert channels on these paradigms.

3.1 The Principle of Least Authority

The principle of least authority states that a process should be given only the minimum access rights required to accomplish its job. In terms of the Password-Capability System this means that a process should be given access only to those capabilities it requires to accomplish its job.

Suppose an email client is installed on a company computer. We may require the client to be able to access locally stored emails and to send emails across the network, etc. However, the process should not be able to access other files on the local system, such as internal company documents. Thus if the client is compromised, it will not be possible to make it email the internal documents to a remote system. This can be achieved in the Password-Capability System by preventing the email client from having access to capabilities for the documents.

On conventional operating systems it is not generally feasible to adhere to the principle of least authority. For instance, on Windows and Unix a user typically has a single login and all his applications run within the single security level of that login. This leaves such systems susceptible to widespread damage should the domain be compromised. In our example the compromised email client would be able to access and leak the internal documents.

3.2 Process Confinement

The Password-Capability System supports a flexible mechanism for process confinement. We say a process is confined if it is unable to communicate information to any other process (except those confined in the same way), unless explicitly authorised. The confinement mechanism is now described, as is an example of its use to solve a real-world problem not readily solvable in conventional systems.

The Confinement Mechanism. Each process in the Password-Capability System has an associated bit-string equal in length to a capability password, called a *lockword*, which it cannot read. Any process having a suitable capability can modify the lockword of another process by xor with an arbitrary value, using *apply_lock*(). Whenever a process tries to use some capability which would enable it to communicate information (called an *alter* capability), the system first decrypts the capability's password by xor with the process's lockword before checking the capability's validity.

Initially a process has its lockword set to that of its creator. By default it is zero, in which case the xor has no effect and the process can use any capability it possesses. To confine a process X, some process Y can modify X's lockword by a chosen value, L, using *apply_lock*(). Now when X tries to use an alter capability, the capability password will be modified by xor with the lockword and the result will not be a valid capability, so the call will fail. X will still be able to use any *non-alter* capabilities it possesses.

X could try to guess its lockword. If it were able to do so, it could encrypt and successfully use any alter capability in its possession. This is prevented by having process Y set X's lockword to a random value, making it as difficult to guess as a capability password.

Process Y may need X to use certain alter capabilities. It can authorise alter capabilities for X's use by encrypting their passwords by xor with L (which it knows). These can then be passed to X and when used by X will be correctly decrypted by the system. Obviously process Y must not pass X an encrypted capability if X knows the unencrypted capability, since it will be able to derive its lockword from the pair.

This mechanism can also solve transitive confinement as formulated by Lampson [17]. He requires that any untrusted processes called by a confined process must themselves be confined. By defining the rights needed to call a process as alter rights, we ensure that the confined process X cannot immediately use them to call some untrusted process Z. Instead X must request that Y authorises a capability to enable it to call Z. Before it does this, Y will take one of two actions.

Either Y will confine the untrusted process Z under the same lockword as X, or it will determine that Z is trusted (according to criteria of its choosing) and leave it unconfined. Additionally, if a confined process creates a new process, this new process is automatically confined with the same lockword as its creator. Thus, the mechanism satisfies Lampson's requirement for transitive confinement.

Confinement Example. Suppose a company wishes to use a third-party word processor to modify some sensitive internal documents with out the word processor being able to leak the documents. This can be achieved through confinement.

In the Password-Capability System, we create a process to execute the word processor code. Before we begin execution of the code, we confine the process. Now the word processor will not be able to use any alter capabilities embedded in its code, so it will be unable to communicate information back to the third-party. To allow the word processor to modify the internal documents, we can authorise alter capabilities to the documents for it to use.

It is not clear how the same problem could be solved in conventional systems such as Windows and Unix. The third-party software would have full access to everything the logged-in user has, and therefore to any network connection the user does, etc. Without putting additional mechanisms in place it will be difficult to ensure the documents are not leaked.

4 The Covert Channel Problem

In this section we introduce *covert channels*. We first give a definition of covert channels as they relate to the Password-Capability System. We show that they could undermine the two security paradigms presented in the previous section. This gives us sufficient motivation to carry out a systematic search for covert channels, which we do in the next section.

4.1 Definition

The covert channel problem was first identified by Lampson [17]. Since then several different definitions of covert channels have been proposed [18]. For our purposes we follow the lead of [18] and use the definition in [19] that a covert channel is a communication channel that allows a process to transfer information in a manner that violates the system's security policy. We will now outline the application of this definition to our system.

If some means exists which allows a process, A, to communicate information to some other process, B, then we say a *communication channel* exists from A to B. If processes within the system are subject to a security policy, restrictions may be placed on their ability to communicate with one another. For instance, a security policy may specify that process A is not permitted to communicate information to process B. The means by which communication channels exist between two processes can be subtle and, if care is not taken, actions taken to enforce the restrictions may miss channels. We call any communication channel which exists in violation of the current security policy a *covert channel*.

4.2 Violating the Security Paradigms

Let us combine the two examples given of uses of the security paradigms. Suppose on a company computer a user is running an email client and using a word processor to work on sensitive company documents. The email client is restricted under the principle of least authority and is unable to access the sensitive documents. The word processor is restricted under process confinement and is unable to communicate with any other process (and may only write information to the sensitive documents). Given this the user may be confident that the email client and word processor cannot conspire to leak the documents.

However, suppose the word processor can, in some subtle way, communicate information to the email client. This would constitute a covert channel and would allow information about the documents to be passed to the email client which could then email it to a third-party. The principle of least authority is undermined, since the email client is able to learn information about the contents of the documents despite never having been given access rights to allow it to do so. Process confinement is also undermined, since the word processor has communicated information to email client without having been authorised.

Of course, the covert channel arises from a failure to correctly implement the two security policies, not from failings in the policies themselves. If we are able to eliminate all covert channels, then the policies will be correctly implemented and it will not be possible to violate them as described. The identification and elimination of covert channels are the topics of the next sections.

5 Communication Channel Identification

In this section we attempt to enumerate all possible communication channels in the Password-Capability System. This will allow us to determine the presence of covert channels in the two security policies outlined previously since, by our definition, covert channels are simply communications channels which exist in violation of the given security policy.

5.1 Scope

This analysis addresses communication channels which can arise in the formal model of the system. We also address channels arising from resource limitations. To this end, we consider the effects of limited storage and processor availability.

Our analysis seeks to identify communication channels between processes. We do not need to identify indirect channels separately; an indirect channel is simply a chain of direct channels.

We do not consider channels which arise from other implementation specific causes. We regard this as future work, to be carried out when the exact details of the implementation in question are known.

5.2 Communication Channels

Communication channels exist in the model if some variable exists which can be written to by some process A and read by some other process B. The simplicity of the formal model makes it easy to enumerate all such variables. We will now examine each variable and attempt to identify the ways in which it can be written and read.

The Virtual Memory, M: The virtual memory is defined by the set of extant volumes which compose it. If a process could create or destroy volumes then it may be able to transmit information through the variable M. No such operation exists for processes, so M cannot be used to form a communication channel between two processes.

Volumes, V: The state of a volume is defined by the set of objects which reside on it. Some process, A, can encode information in the state of a volume by creating new objects, deleting existing ones or by preventing their deletion.

If process A creates a new object, another process, B, can detect this only if it has a capability to the object. Since only process A initially has a capability to the object and it is infeasible for B to guess it, no communication channel can be established.

Process A can delete an object by either deleting its master capability or by withdrawing money from the object, such that the object can no longer pay its rental charge. If process B has a capability to the object, then it can detect the destruction of the object by the failure of an attempt to use the capability. Hence a communication channel can be established.

Process A can prevent the deletion of an existing object by depositing money into the object to cover its rent. The ongoing existence of the object can be detected by process B if it has a capability for it. Hence a communication channel can be established.

Objects, O: Process A can encode information in the existence or otherwise of objects. Process B can discover partial information about these objects (their collective size) by creating further objects of known size until all available storage space on the volume is exhausted. Hence a channel can be established.

Alternatively process A can exhaust all available storage or not, at certain times. Process B can detect this by attempting to create new objects. Hence a communication channel can be established.

Data Area, data: Process A can modify the state of the data area either by altering the data stored in the area or by changing the size of the data area.

It can alter the data stored in the data area using the *write*() command. Process B can read this change using the *read*() command. Hence a communication channel can be established.

Process A can alternatively alter the size of the data area using the *extend*() command. Process B can read this change in three ways. It can call $cap\_info()$

with a capability having appropriate rights to discover the size of the object. It can determine the size from the amount of rent being charged by checking the amount of money left in the object (using *cap_info*() or *withdraw*()), or by timing how long it takes for the object to be deleted (and checking deletion by trying to use a capability for the object). Process B can also detect the size of the data area by filling up all available storage space on the object's volume by creating new objects of known size. Doing so before and after the change in size of the object's data area would yield the change in size of the data area. Hence a communication channel can be established.

Channels which arise from the use of the object as a process, utilise the fields into which the data area is separated (lockword, cashword, etc.). Each of these fields will be treated individually.

Process Type, type: The *type* field of a process specifies the type of processor the process can use. Process A can set the *type* field using *revive*() on a new or terminated process. Process B can deduce the contents of the type field by determining which processors the process will or will not run on. Hence a communication channel can be established.

Process Status, status: Process A can modify a process's *status* field by calling any of *send*(), *wait*(), *suspend*(), *resume*() and *revive*(). Process B can read the *status* field by calling *process_info*(). Alternatively, B can deduce information about the status of a process by observing whether or not it will run on available processors. Hence a communication channel can be established.

Process Program Counter, pccap and pc: Process A can set some process's program counter using *revive*(), or it can control its advance by selective use of *suspend*() or *wait*(). Process B cannot observe the program counter directly. It could attempt to deduce it by monitoring process A's behaviour. However, this requires the existence of some other channel through which to carry out the observation. We are only interested in communications channels which can exist on their own. Hence no communication channel can be established.

Process Cashword, cash: Process A can modify a process's cashword in a number of ways. It can vary when the process is run, using *resume*(), *send*(), *suspend*() and *wait*(), to alter the rate at which payment is taken from the cashword. It can use *withdraw*() or *deposit*() to vary its own cashword. It can use *send*() to transfer cash from its cashword to that of another process. It can use *receive*() to augment its cashword with money in its mailbox. It can use *revive*() to give cash to another process. It can vary the operations it carries out to vary how much it is charged. Process B can read a process's cashword using *process_info*(). Hence a communication channel can be established.

Process Lockword, lock: Process A can modify a process's lockword by using *apply_lock*(). Process B can read from its lockword by trying to use a capability with its password xored by a guessed value of the lockword. If Process B guesses the lockword correctly the capability will work. If processes A and B have previously established a code giving meaning to a number of possible lockword values, then a communication channel can be established.

Process Mailbox, mailbox: Process A can modify the state of a mailbox using *send*(). Process B can read from its mailbox using *receive*(). Hence a communication channel can be established. There is no way for a process to directly read from the mailbox of another process. However, the process can detect whether or not the recipient's mailbox is full, by the success of a send call. A third process could selectively fill the mailbox of the recipient to convey information to the first. Hence a channel can also be established this way.

Capability Tree, caps: Process A can modify an object's capability tree by deleting capabilities, using *delete_cap*() or *rename_obj*(). Process B can detect such modifications by attempting use capabilities to determine whether they have been revoked. Hence a communication channel can be established.

Capability Rights, rights: Process A can encode information in the access rights of a capability only at the time the capability is created. However, unless some other communication channel already exists, there is no way for process B to learn this capability other than guessing it, which is expected to be infeasible. Hence no communication channel can be established.

Capability Window, start and size: Process A can a capability's window using either *extend*() or *contract*(). Process B can detect this using *cap_info*(), or by observing the effect of a call which makes use of the window: *read*(), *write*(), *revive*() or *derive_cap*(). Hence a communications channel can be established.

Capability Parent, parent: Process A can select the parent from which a capability is to be created and may therefore be able to encode information in the *parent* field of the capability. However, process B cannot access this capability unless it can guess the capability password, which is expected to be infeasible. Hence no communication channel can be established.

Capability Moneyword, money: Process A can modify the moneyword of a capability by using *withdraw*() or *deposit*() on it (or one of its derivatives). Process B can observe this using either *cap_info*(), or by using *withdraw*() to determine how much money can be extracted from the capability. Hence a communication channel can be established.

6 Covert Channel Elimination

In this section we return to our two security paradigms and assess the extent to which they are affected by covert channels. We show how to eliminate all known covert channels from these policies.

6.1 The Principle of Least Authority

To correctly support the principle of least authority, we require that no process can communicate information to another process, unless it is given a capability to do so. The only legitimate means of inter-process communication are the use of *send*() to pass a message directly, and the use of *read*() and *write*() to communicate through a shared object. Unless two processes, A and B, have been given capabilities permitting either of these legitimate forms of communication, they must not be able to communicate. Obviously this means that no covert channels should exist between them.

Suppose we wish a process to enforce a security policy by distributing capabilities based on the principle of least authority. How do we ensure that it does not inadvertantly create covert channels? We do so by having it check, before it distributes each capability, that the creation of covert channels will not result.

To this end, the must keep a record of all capabilities it has already distributed. When about to distribute a new capability, it can check each of the communications channels identified in the last section in turn and determine whether adding the new capability to the set of previously distributed capabilities would open up any covert channels.

This method will prevent all covert channels which rely on the use of capabilities. Unfortunately, certain covert channels operate using only *make_obj*(), for which no capability is required. To rectify this situation, we propose that *make_obj*() be modified to require a capability. We propose that capabilities for extant objects on a volume can have a new *make_obj* right, allowing a new object to be created on the same volume. Making this change allows all covert channels to be eliminated from the Password-Capability System with respect to the principle of least authority.

6.2 Process Confinement

To ensure that a process confined under the Password-Capability System's confinement mechanism cannot communicate, unless authorised, we must again ensure that no covert channels exist. The confinement mechanism allows us to control the use of any capability which could be used to establish a communications channel. We designate any such capability as an alter capability. Hence, an alter capability is any capability conferring an access right which allows the capability to be used to communicate information. Using the list of communication channels identified earlier we can determine which rights can in fact be used to establish a communication channel and declare them as alter rights. Note that we assume, as just proposed, that *make_obj*() requires a capability with the

corresponding access right. This set of alter rights, $ALTER$, is therefore:

$$ALTER = \{delete\_cap, withdraw, deposit, write, extend, \\ send, wait, suspend, resume, revive, \\ apply\_lock, rename\_obj, contract, make\_obj\}.$$

This allows us to eliminate all covert channels except one. A process can modify its cashword by varying the operations it carries out, to vary how much it is charged. Another process able to read the cashword can learn information encoded in it. Since the process does not need to use a capability for this, this channel cannot be eliminated by locking alter capabilities. Instead we suggest that when one process confines another, it does not distribute capabilities having the right to read the confined process's cashword (i.e. having the cap_info right). This will eliminate the channel. Thus we can ensure that the confinement mechanism is free from all known covert channels.

When authorising an alter capability for use by some confined process, it is important to establish that doing so does not give rise to any unintended communication channels. This can be done by considering the list of potential communications channels and the means by which they arise.

7 Discussion and Conclusion

The Password-Capability System offers an alternative to conventional operating systems. It has a strong security model which, we demonstrated, can be used to solve problems that conventional systems cannot. We gave as an example the secure use of untrusted third-party software, such as email clients and word-processors. We formulated these problems in terms of the underlying security paradigms: the principle of least authority and process confinement.

The issue of covert channels must be addressed if we are to have confidence in the Password-Capability System's support of the paradigms. We have attempted to enumerate all covert channels in the system. We have shown how to eliminate these channels from the paradigms. To do so, it was necessary to modify the Password-Capability System slightly, introducing an access right which controls the creation of new objects.

We cannot guarantee that our analysis has identified all covert channels within its scope. However, during the analysis we have discovered and eliminated covert channels not previously handled by the system (hence the need to modify the system). There are now fewer or no covert channels left in the system for exploitation. The Password-Capability System is more secure as a result.

We have analysed the formal system model and some common implementation issues (i.e. resource limitations). Any implementation of the Password-Capability System should be analysed further to ensure that no covert channels result from the details of that implementation. For now we conclude that the Password-Capability System is a secure operating system model supporting two powerful security paradigms which are free from known covert channels.

References

1. Dennis, J.B., Van Horn, E.C.: Programming semantics for multiprogrammed computations. Communications of the ACM **9** (1966) 143–155
2. Myers, G.J., Buckingham, B.R.S.: A hardware implementation of capability-based addressing. ACM SIGARCH Computer Architecture News **8** (1980) 12–24
3. Keedy, J.L., Espenlaub, K., Hellman, R., Pose, R.D.: SPEEDOS: How to achieve high security and understand it. In: Proceedings of CERT Conf. 2000, Omaha, Nebraska, USA (2000)
4. Abramson, D.A., Rosenberg, J.: The microarchitecture of a capability-based computer. In: Proceedings of the 19th annual workshop on Microprogramming, New York, USA (1986) 138–145
5. Cohen, E., Jefferson, D.: Protection in the Hydra operating system. In: Proceedings of the Fifth ACM Symposium on Operating System Principles, ACM Press, New York, NY, USA (1975) 141–160
6. A.K.Jones: Capability architecture revisited. Operating Systems Review **14** (1980) 33–35
7. Mossop, D., Pose, R.: Semantics of the Password-Capability System. In: Proceedings of the IADIS International Conference, Applied Computing 2005. Volume 1. (2005) 121–128
8. Castro, M.D.: The Walnut Kernel: A Password-Capability Based Operating System. PhD thesis, Monash University (1996)
9. Wallace, C.S., Pose, R.D.: Charging in a secure environment. Security and Persistence, Springer-Verlag (1990) 85–97
10. Anderson, M., Wallace, C.S.: Some comments on the implementation of capabilities. The Australian Computer Journal **20** (1988) 122–130
11. Anderson, M., Pose, R.D., Wallace, C.S.: A password-capability system. The Computer Journal **29** (1986) 1–8
12. Heiser, G., Elphinstone, K., Vochteloo, J., Russell, S., Liedtke, J.: The Mungi single-address-space operating system. Software Practice and Experience **28** (1998) 901–928
13. Vochteloo, J.: Design, Implementation and Performance of Protection in the Mungi Single-Address-Space Operating System. PhD thesis, University of NSW, Sydney 2052, Australia (1998)
14. Vochteloo, J., Elphinstone, K., Russell, S., Heiser, G.: Protection domain extensions in Mungi. In: Proceedings of the 5th IEEE International Workshop on Object Orientation in Operating Systems, Seattle, WA, USA (1996)
15. Vochteloo, J., Russell, S., Heiser, G.: Capability-based protection in the Mungi operating system. In: Proceedings of the 3rd IEEE International Workshop on Object Orientation in Operating Systems, Asheville, NC, USA (1993)
16. Chase, J.S., Baker-Harvey, M., Levy, H.M., Lazowska, E.D.: Opal: A single address space system for 64-bit architectures. In: Proceedings of the Third Workshop on Workstation Operating Systems, ACM Press, New York, NY, USA (1992) 80–85
17. Lampson, B.W.: A note on the confinement problem. Communications of the ACM **16** (1973) 613–615
18. N.C.S.C.: A guide to understanding covert channel analysis of trusted systems. Technical Report NCSC-TG-030, National Computer Security Center (1993)
19. Tsai, C.R., V.D.Gligor, Chandersekaran, C.: A formal method for the identification of covert storage channels in source code. IEEE Transactions on Software Engineering **16** (1990) 569–580

Comparing Low-Level Behavior of SPEC CPU and Java Workloads

Andy Georges, Lieven Eeckhout, and Koen De Bosschere

Department of Electronics and Information Systems (ELIS),
Ghent University, St.-Pietersnieuwstraat 41,
B-9000 Gent, Belgium
{ageorges, leeckhou, kdb}@elis.ugent.be
http://www.elis.ugent.be/paris

Abstract. Java workloads are becoming more prominent on a wide range of computing devices. In contrast to so-called traditional workloads written in C and Fortran, Java workloads are object-oriented and comprise a virtual machine. The latter includes a runtime environment with garbage collection, Just-In-Time (JIT) compilation, etc. As such, Java workloads potentially have different execution characteristics from traditional C or Fortran workloads. In this paper, we make a thorough comparison between SPEC CPU and Java workloads using statistical data analysis techniques and performance counters on an AMD Duron platform. In our experimental setup we use four virtual machines for the Java workloads running SPECjvm98, SPECjbb2000 and Java Grande. Our main conclusion is that Java workloads are significantly different from SPEC CPU and that the execution characteristics for which Java workloads differ from SPEC CPU, is subjective to the virtual machine; we can make a distinction between mixed-mode and compilation-only virtual machines.

1 Introduction

Performance evaluation of a microprocessor, during and after design time, is a time-consuming process, involving a large number of benchmarks. It is paramount that the benchmarks used for performance analysis are representative for the workloads that will actually be run on the hardware. One particular CPU-intensive benchmark suite offering 26 real-life applications is the Standard Performance Evaluation Corporation's (SPEC) CPU suite.

However, with the recent advent of Java applications on various computing devices, the point can be raised whether Java workloads should be taken into account next to SPEC CPU during performance analysis. To address this issue a number of questions need to answered. How different are Java workloads from SPEC CPU workloads? If they are, can we pinpoint the main reasons for this? Do the differences depend on the VM we use in the experiments?

To answer these questions, we have done extensive measurements using the performance counters on an AMD K7 microprocessor, for a large number of benchmarks: SPEC CPU2000 and four Java virtual machines (VMs) running

T. Srikanthan et al. (Eds.): ACSAC 2005, LNCS 3740, pp. 669–679, 2005.

SPECjvm98, Java Grande Forum, and SPECjbb2000. We use rigorous statistical techniques, such as Principal Components Analysis, Cluster Analysis, the Hotelling T^2-test and the *t*-test, to determine which execution characteristics are significantly different between SPEC CPU and Java workloads. While a substantial amount of previous work [1,7,9,10,11,13] exists on characterizing and comparing Java workload behavior versus SPEC CPU, we deem it worth to redo some of that work for the following reasons. First, previous work was mostly done on a SUN (SPARC) platform; this paper uses the IA-32 ISA, a far more popular platform. Second, since we use the IA-32, we are capable of running multiple virtual machines whereas previous work typically used only one or two virtual machines. In addition, the virtual machines used in this paper are today's state-of-the-art. Third, this paper uses performance counters on native hardware to gather execution characteristics. Previous work typically used simulation which is both slower and less accurate. Fourth, the use of performance counters allows the analysis of long running Java applications such as SPECjvm98 with the `s100` input set and SPECjbb2000. Previous work typically used SPECjvm98 with the `s1` and `s10` input sets to limit simulation time. And finally, we use rigorous statistical analysis techniques to compare Java versus SPEC CPU workloads; previous work typically just compared average values.

We conclude that Java workloads and SPEC CPU workloads are significantly different from each other. Java workloads have significantly less L1 D-cache misses, significantly more L2 I-TLB misses and significantly more mispredicted function returns. When looking at individual virtual machines we make the interesting and previously unnoticed observation that a clear distinction can be made between mixed-mode (both interpretation and compilation) and compilation-only virtual machines. For example, the mixed-mode virtual machines have significantly more mispredicted indirect branches than SPEC CPU, whereas compilation-only VMs do not. Throughout the discussion, we also show that several observations that were made in previous work do not hold for long running Java workloads on IA-32 virtual machines.

The remainder of this paper is organized as follows. After a brief overview of the experimental setup, we touch upon the statistical techniques we used (section 3). In section 4 we present and discuss the results of our experiments. Finally, we conclude in section 5.

2 Experimental Setup

2.1 Measuring with Performance Counters

Our experiments have been conducted on the AMD Duron, a member of the AMD K7 family, implementing the IA-32 architecture. It is a superscalar out-of-order microprocessor, with a 10-stage pipelined micro-architecture [3]. The processor we have used in our experiments is clocked at 1GHz, with a bus clock frequency of 100MHz. The machine has 1 GiB of main memory.

Modern microprocessors, such as the AMD Duron, are equipped with a set of microprocessor-specific registers that contain a number of so called *performance*

Table 1. The 34 performance characteristics obtained from the performance counters on the AMD Duron

component	measured characteristics
general	clock cycles, retired x86 instructions, retired operations, retired branches, retired taken branches, retired far control instructions, retired near return instructions
processor front-end	L1 I-cache fetches, L1 I-cache misses, L2 instruction fetches, instruction fetches from memory, L1 I-TLB misses (but L2 I-TLB hits), L1 and L2 I-TLB misses, fetch unit stall cycles
branch prediction	retired mispredicted branches, retired mispredicted taken branches, retired mispredicted near return instructions, mispredicted branches due to address miscompare, return address stack hits, return address stack overflows
processor core	dispatch stall cycles, integer control unit (ICU) full stall cycles, reservation station full stall cycles, floating-point unit (FPU) full stall cycles, L1-cache load-store unit (LSU) full stall cycles, L2 and memory load-store unit (LSU) full stall cycles
data cache	L1 data cache accesses, L1 data cache misses, refills from the L2 cache, refills from main memory, writebacks, L1 D-TLB misses (but L2 D-TLB hits), L1 and L2 D-TLB misses
system bus	number of memory requests as seen on the bus

counter registers. These registers count occurrences of certain events in the processor. Unlike instrumentation or simulation, the use of performance counters allows measuring events at the speed of a native execution. Additionally, these measurements are performed on actual hardware instead of a software model, thus removing inaccuracies due to the higher abstraction level of most architectural simulation models [2]. We used the perfctr[1] package to read the contents of the performance counters, on a Gentoo[2] Linux distribution, running the Linux kernel version 2.4.20.

For our analysis, we have measured 34 performance counters, see Table 1. Each one of these is divided by the number of elapsed clock cycles while executing the workload, effectively yielding 33 normalized *performance characteristics*. This way, we obtain measurements that count the number of events per unit of time. To eliminate inaccuracies in our measurements, we ran all experiments four times, using the arithmetic average of each measured performance counter in our analysis.

2.2 Workloads

We have used a total of 99 workloads: 42 from SPEC CPU and 57 Java workloads. We have used the entire SPEC CPU2000[3] suite except for 178.galgel and 175.vpr; the former because it did not compile, the latter because it did

[1] http://user.it.uu.se/~mikpe/linux/perfctr/
[2] http://www.gentoo.org
[3] http://www.spec.org/cpu2000

not execute properly. SPEC CPU comes with two sets of benchmarks: integer benchmarks (SPECint) and floating-point benchmarks (SPECfp). The benchmarks have been compiled using the Intel C and Fortran compilers, version 7.1. We have used the `-O3 -ip -ipo -axM` optimization options. For the Java workloads, we use multiple Java virtual machines, because the impact of a JVM on the execution behavior can be significant [4]. We have used the following virtual machines: (i) SUN JRE 1.4.1 in server mode using non-incremental generational garbage collection (GC), (ii) Jikes RVM 2.2.0 in adaptive mode using non-generational copying GC, (iii) JRockit 8.1 with generational copying GC, and (iv) Kaffe 1.1.0 with a conservative mark-and-sweep GC. Two VMs use a mixed-mode scheme, i.e. interpretation and JIT compilation, (Kaffe and SUN), while the other two only use compilation (Jikes and JRockit). The Java benchmarks were taken from the SPECjvm98[4] suite with the `s100` input set, the 'sequential large scale' applications from the Java Grande Forum[5] benchmark set, and the SPECjbb2000 suite with 2, 4, and 8 warehouses. We have used a heap size of 64MiB for all Java applications, except for SPECjbb2000, for which we used a 512MiB heap. We were unable to complete a run for SPECjbb2000 with JikesRVM.

3 Statistical Techniques

The experiments we conducted yield a large amount of information, namely a 99×33 matrix (99 workloads and 33 performance characteristics). Clearly, trying to understand the differences between workloads based on this is quite infeasible, especially since several characteristics are correlated with each other. Conceptually, the workload space can be viewed as a 33-dimensional space, spanned by the performance characteristics. Principal Components Analysis (PCA) [8] is a statistical technique that can be used to reduce the dimensionality of the data, from $n = 33$ to $n = 6$, while still retaining a large portion of the information or variance observed in the data set. For this, new spanning dimensions are computed as linear combinations of the original characteristics. In this new space, which is called the PCA space, the dimensions show no correlation, making the Euclidean distance reliable for determining the (dis)similarities between workloads [4,5,6].

To determine which characteristics are significantly different between the two groups of workloads (Java vs. SPEC CPU), we use two statistical tests: the Hotelling T^2-test for comparing multivariate mean vectors and the t-test for comparing univariate means. For the t-test, to conclude that there is a significant difference between both means, at the 90% confidence level, the p-level should be smaller than 0.1.

In a n-dimensional space, with $n > 2$, if is often difficult to obtain a clear understanding of the (dis)similarities between the workloads residing in the space. To overcome this, we use cluster analysis. This technique is aimed at clustering

[4] `http://www.spec.org/jvm98`
[5] `http://www.javagrande.org`

data points, where the most closely related points will be clustered first. We have chosen to perform cluster analysis on the data points residing in the principal components space. The main reason for this choice is that there is no correlation between the axis spanning the principal components space. This means that the Euclidean distance can be relied on as a metric to measure the dissimilarity between two workloads. The clustering technique used in our analysis is the complete linkage strategy, using the greatest distance between any members of two clusters as a distance metric. Initially, each workload forms a separate cluster. Iteratively, clusters that have the smallest (complete) distance between their members are joined to form a larger cluster until the algorithm ends with a single cluster. A graphical representation of such a clustering is called a dendrogram.

4 Results

4.1 PCA Space

Figure 1 shows the PCA space in its first two dimensions or principal components PC_1 and PC_2. These account for 52.14% of the total variance. Essentially, larger values for PC_1 indicate higher IPC and poorer branch prediction accuracy, smaller values indicate more stalls and poorer D-cache behavior. For PC_2, larger values indicate more D-cache accesses, while smaller values indicate poorer I-cache behavior and poorer branch prediction accuracy. We do not show the

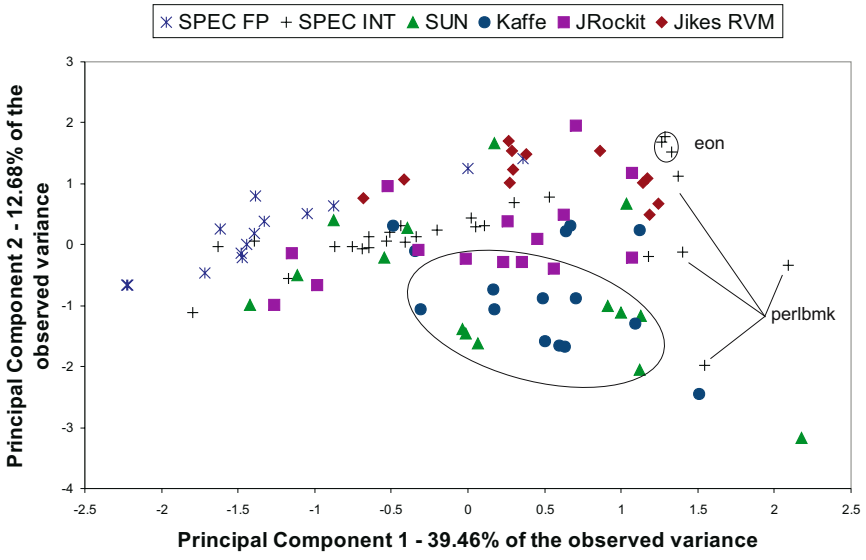


Fig. 1. Scatter plots for the Java and SPEC CPU2000 set as a function of the first two principal components

other PC_i since SPEC CPU and Java workloads do not show as large dissimilarities in those plots. From Figure 1, we can make the following observations.

The SPECfp workloads can mostly be found in a quite small region in the PC_1/PC_2 scatter plot. SPECfp shows better I-cache behavior than SPECint, due to higher code locality. The smaller IPC can be explained by the fact that they have more stalls due to full floating point units.

The data points for both JRockit and Jikes lie relatively closer to SPEC CPU than the data points for SUN and Kaffe. A possible explanation might be that JRockit and Jikes do not interpret the Java bytecode, but instead use an compilation-only scheme. A large part of the Java workloads run on Kaffe and SUN occupy a region that is quite apart from SPEC CPU, see the ellipse in Figure 1. This interesting observation, i.e. the (dis)similarity between the mixed-mode VMs and the compilation-only VMs, will be studied in more detail in the next section.

4.2 Hotelling T^2-Test and *t*-Test

Using the Hotelling T^2-test, we conclude that there is a statistically significant difference in the characteristic mean vectors (built up by the 33 performance characteristics) for Java and SPECint, as well for Java and SPECfp. In both cases the null hypothesis, i.e. the equality of the mean vectors, is rejected at a 90% significance level.

We have then used the *t*-test to compare the Java workloads with SPEC CPU for each individual performance characteristic. For Java vs. SPECfp, we have dissimilarities for almost all characteristics, thus showing that Java workloads do behave significantly different from classical floating-point workloads. In the remainder of this analysis, we have thus limited our study to comparing Java workloads versus the SPECint workloads.

Table 2 compares SPEC CPU versus the Java workloads for several interesting performance characteristics. In Table 2, we make a distinction between the mixed-mode VMs (SUN and Kaffe), and the compilation-only VMs (JRockit and Jikes). Here are the most important conclusions:

- We observe that Java workloads exhibit significantly better D-cache behaviour than SPEC CPU, which confirms the results of Radhakrishnan *et al.* in [13]. However, we do not find better I-cache behaviour, in contrast to [13]. We also see that there are significantly more refills to the I-cache from the main memory for the mixed-mode VMs when compared to SPECint, which is not the case for the compilation-only VMs.
- All Java workloads have a larger L2 I-TLB miss rate, but a smaller L1 D-TLB miss rate on the compilation-only VMs. The larger I-TLB miss rate can be explained by the fact that code is spread out over more pages, for both mixed-mode VMs as for compilation-only VMs. Moreover, the execution swaps between application code and VM code, thereby increasing the number of pages that must be accessed. In [11], Tao *et al.* state that Java workloads also show bad D-TLB behaviour. According to our experiments however, we observe no worse behaviour than for SPECint.

Table 2. Comparing SPEC CPU vs. Java workloads using the *t*-test. The 'mean' columns show the number of events per 1,000 cycles; the 'p' columns show the *p*-level of the *t*-test for the comparison.

	SPEC	Sun + Kaffe		Jikes + JRockit		All VM's	
characteristic	mean	mean	p	mean	p	mean	p
D-cache misses	4.24	2.69	0.015	2.33	0.005	2.52	< .001
D-cache memory refills	1.34	1.25	0.738	0.95	0.178	1.11	0.335
D-TLB L2 hits	3.16	2.38	0.302	1.65	0.038	2.04	0.061
D-TLB L2 misses	0.32	0.45	0.331	0.30	0.886	0.38	0.611
I-cache misses	0.92	1.29	0.346	0.75	0.654	1.04	0.700
I-cache system refills	0.11	0.29	0.020	0.16	0.345	0.23	0.043
I-TLB L2 hits	1.74	1.92	0.776	2.86	0.171	2.37	0.325
I-TLB L2 misses	1.1E-3	0.05	< .001	0.02	< .001	0.03	< .001
RAS[6] hits	8.47	9.04	0.767	13.86	0.027	11.32	0.136
RAS overflows	9.8E-3	2.48	< .001	0.38	< .001	1.49	< .001
branches	100.65	78.95	0.039	84.92	0.190	81.78	0.036
branches mispredicted	6.56	7.79	0.332	4.09	0.010	6.04	0.607
far control transfers	1.8E-3	0.01	0.003	0.08	0.030	0.04	0.097
branch targets mispredicted	1.65	5.16	0.001	1.00	0.309	3.19	0.065
near returns	6.97	6.72	0.876	12.46	0.012	9.44	0.148
near returns mispredicted	0.25	3.15	< .001	1.20	< .001	2.23	< .001

- The compilation-only VMs show dissimilar branch prediction behaviour from SPECint, whereas the mixed-mode VMs do not. The former have significantly fewer mispredictions. In general, Java workloads have fewer branches than SPECint, in contrast to the results obtained by Li *et al.* in [10].
- Our results show that the BTB-behaviour of Java applications run on the mixed-mode VMs is far worse than for SPECint. This is most likely due to the interpreter, which employs a large switch statement, resulting in many target mispredictions [9]. Similar results for an interpreting VM were obtained by Hsieh *et al.* in [7]. For the compilation-only VMs we see no significant difference with SPECint. Our results contrast with the conclusions made by Chow *et al.* in [1] which states that Java workloads have more indirect branches, but less branch targets, resulting in more or less similar behaviour between Java and SPEC CPU95.
- Java workloads also exhibit more far control transfers, and more near return mispredictions. This can be explained by the fact that Java workloads have more function calls, resulting in a deeper call tree. Also, the OO-framework gives rise to virtual method calls, making target prediction more difficult. This confirms the findings of Li *et. al.* in [10]

4.3 Cluster Analysis

On the data we obtained in section 4.1 from the principal components analysis, we can now apply cluster analysis. As discussed in section 3, a dendrogram then

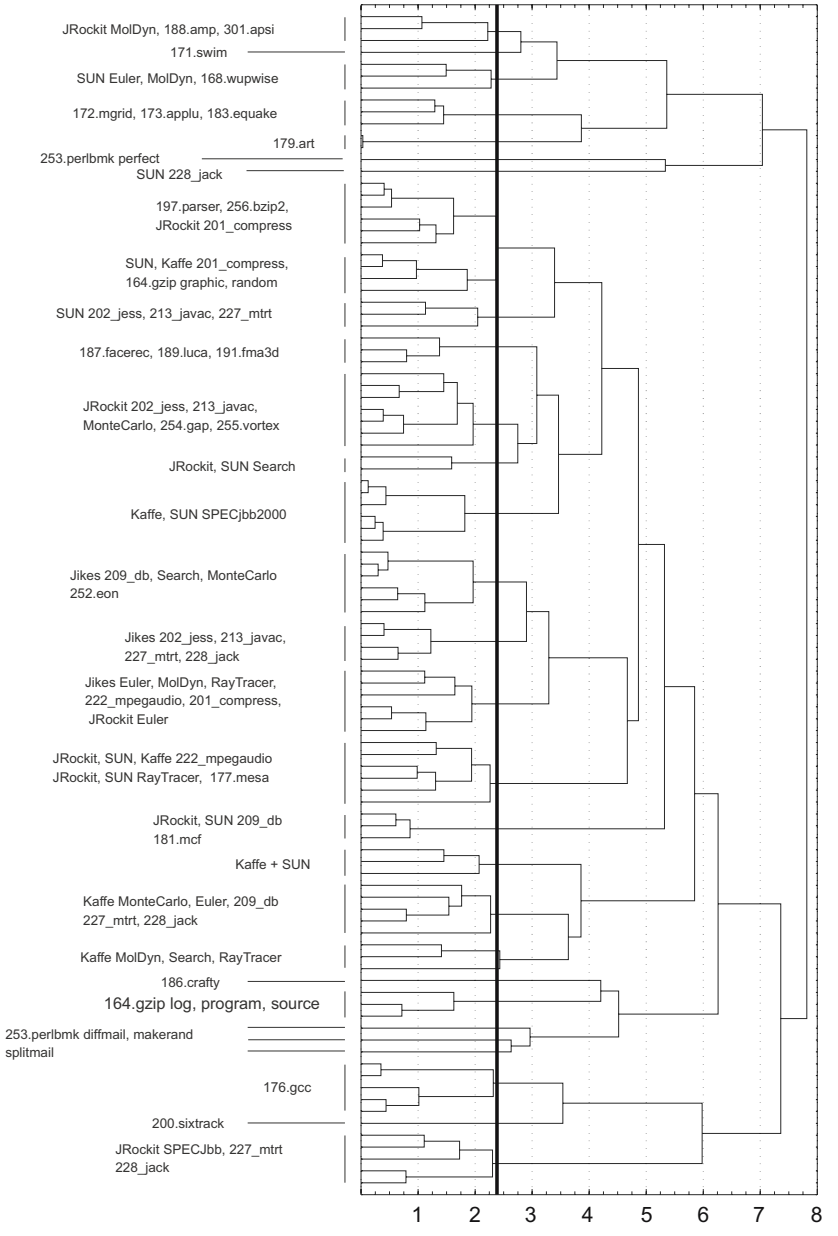


Fig. 2. Dendrogram for Java and the SPEC CPU benchmarks, using the average linking strategy. We identify 30 clusters based on the critical linkage distance of 2.34 (shown by the thick vertical line).

Table 3. Selecting a representative workload subset. The representative workloads per clusters are shown in bold.

cluster	number	benchmarks
1	3	JRockit MolDyn, **188.amp**, 301.apsi
2	1	**171.swim**
3	3	**SUN** {Euler, **MolDyn**}, 168.wupwise
4	3	172.mgrid, **173.applu**, 183.equake
5	2	**179.art** (both inputs)
6	1	**253.perlbmk.perfect**
7	1	**SUN 228_jack**
8	6	197.parser, **256.bzip2**.{**input.source**, input.graphic, input.program}, JRockit 201_compress
9	4	{SUN, Kaffe} 201_compress, **164.gzip**.{**graphic**, random}
10	3	**SUN** {202_jess, **213_javac**, 227_mtrt}
11	3	187.facerec, **189.lucas**, 191.fma3d
12	7	JRockit {202_jess, 213_javac, Montecarlo}, 254.gap, **255.vortex**.**lendian**{1,2,3}
13	2	{**JRockit**, SUN} **Search**
14	6	{Kaffe, **SUN**} **SPECjbb200** (all inputs)
15	6	Jikes {209_db, Search, Montecarlo}, **252.eon** {**kajiya**, cook, rushmeier}
16	4	**Jikes** {202_jess, 213_javac, **227_mtrt**, 228_jack}
17	6	**Jikes** {Euler, MolDyn, Raytracer, **222_mpegaudio**, 201_compress}, JRockit Euler
18	6	{JRockit, SUN, Kaffe} 222_mpegaudio, {**JRockit**, SUN} **RayTracer**, 177.mesa
19	3	{JRockit, **SUN**} **209_db**, 181.mcf
20	3	**Kaffe** {202_jess, **213_javac**}, SUN Montecarlo
21	5	**Kaffe** {MonteCarlo, Euler, 209_db, **227_mtrt**, 228_jack}
22	3	**Kaffe** {MolDyn, **Search**, RayTracer}
23	1	**186.crafty**
24	3	**164.gzip** {log, **program**, source}
25	1	**253.perlbmk.diffmail**
26	1	**253.perlbmk.makerand**
27	1	**253.perlbmk.splitmail**
28	5	**176.gcc** (all inputs)
29	1	**200.sixtrack**
30	5	**JRockit** {SPECjbb2000, 227_mtrt, **228_jack**}

gives a visualization of the clustering algorithm. Figure 2 gives the dendrogram we obtain from our dataset. Workloads clustered together with small linkage distances (shown on the X-axis) exhibit similar behavior; workloads clustered together with large linkage distances exhibit dissimilar behavior. This dendrogram can now be used to determine how different SPEC CPU is from the Java workloads. Assuming that we want to select 30 clusters, we need to set the critical linkage distance to 2.34, i.e. the thick vertical line in Figure 2. From this experiment, we can make several interesting observations:

- There are 10 clusters containing only Java workloads. For example, we see that the SUN and Kaffe VMs running SPECjbb are clustered together. Another example is the JRockit cluster found at the bottom of the dendrogram, where SPECjbb and SPECjvm98 applications show similar behaviour.
- There are 12 clusters with only SPEC CPU workloads. Perlbmk is a clear example of this, another cluster contains all the gcc workloads.
- The other clusters contain both Java and SPEC CPU workloads. It is interesting to see that, although the *t*-test showed that for Java Grande workloads and SPEC FP, significant dissimilarities exist between the benchmarks suites as a whole, still a number of clusters contain precisely workloads from these

two categories. For instance, one group contains 188.ammp, 301.apsi and JRockit executing MolDyn. There is another group containing 168.wupwise and SUN executing Euler and MolDyn.

Another application for the cluster analysis, next to gaining insight into workload behavior, is workload composition or benchmark suite subsetting [12,6]. The idea is that the clustering results can be used to select a reduced workload set comprising both SPEC CPU and Java applications that is representative for the entire workload space. The selected workloads per cluster are shown in Table 3 in bold. From each cluster we took the workload closest to the cluster center. This results in 17 SPEC workloads and 13 Java workloads.

In conclusion, we can state that when composing a benchmark suite for CPU design purposes, the benchmark suite should comprise a well chosen mix of SPEC CPU and Java workloads. The reason for this is that Java workloads behave significantly different fro SPEC CPU workloads, as pointed out in this paper.

5 Conclusions

When designing and evaluating microprocessor performance, it is important to understand its workloads. This paper compared SPEC CPU2000 versus Java workloads using rigorous statistical analysis techniques and performance counters on an AMD Duron microprocessor. In this analysis we used four Java virtual machines executing long running Java applications, SPECjvm98 `s100`, SPECjbb2000 and Java Grande Forum. This paper differed from previous work for a number of reasons: the use of performance counters, multiple virtual machines, a different ISA and the use of long running Java applications.

We conclude that Java workloads are significantly different from SPEC CPU for several characteristics, namely less L1 D-cache misses, more L2 I-TLB misses and more mispredicted returns. We also make the interesting observation that mixed-mode virtual machines have different execution characteristics than compilation-only virtual machines. The most striking differences are in the number of L1 D-TLB misses and the number of branch (target) mispredictions. Additionally, we conclude that some observations made in previous work do not hold for long running Java applications on IA-32 virtual machines, for example better I-cache behaviour and worse D-TLB behaviour. Finally, we conclude that a representative benchmark suite for general-purpose CPU design should comprise both SPEC CPU and Java workloads.

Acknowledgments

Andy Georges is supported by the CoDAMoS IWT project. Lieven Eeckhout is a Postdoctoral Fellow of the Fund for Scientific Research—Flanders (Belgium) (F.W.O.—Vlaanderen). This research was also funded by Ghent University.

References

1. K. Chow, A. Wright, and K. Lai. Characterization of Java workloads by principal components analysis and indirect branches. In *Proceedings of the Workshop on Workload Characterization (WWC-1998), held in conjunction with the 31st Annual ACM/IEEE International Symposium on Microarchitecture (MICRO-31)*, pages 11–19, November 1998.
2. R. Desikan, D. Burger, and S. W. Keckler. Measuring experimental error in microprocessor simulation. In *Proceedings of the 28th Annual International Symposium on Computer Architecture (ISCA-28)*, pages 266–277, July 2001.
3. K. Diefendorff. K7 challenges Intel. *Microprocessor Report*, 12(14), October 1998.
4. L. Eeckhout, A. Georges, and K. De Bosschere. How Java programs interact with virtual machines at the microarchitectural level. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA) 2003*, pages 169–186, October 2003.
5. L. Eeckhout, H. Vandierendonck, and K. De Bosschere. Designing computer architecture research workloads. *IEEE Computer*, 36(2):65–71, 2 2003.
6. L. Eeckhout, H. Vandierendonck, and K. De Bosschere. Quantifying the impact of input data sets on program behavior and its applications. *Journal of Instruction-Level Parallelism*, 5:1–33, 2 2003.
7. C.A. Hsieh, M.T. Conte, T.L. Johnson, J.C. Gyllenhaal, and W.W. Hwu. A study of the cache and branch performance issues with running Java on current hardware platforms. In *Proceedings of the 42nd IEEE International Computer Conference (COMPCON 97)*, pages 211–216, feb 1997.
8. R. A. Johnson and D.W. Wichern. *Applied Multivariate Statistical Analysis*. Prentice Hall, fifth edition, 2002.
9. T. Li, R. Bhargava, and L. John. Rehashable btb: an adaptive branch target buffer to improve the target predictability of java code. In *Proceedings of the International Conference on High Performance Computing (HiPC)*, pages 597–608, 2002.
10. T. Li, S. Hu, Y. Luo, L. John, and N. Vijaykrishnan. Understanding control flow transfer and its predictability in Java processing. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*, pages 65–76, 2001.
11. T. Li, L. K. John, V. Narayanan, A. Sivasubramaniam, J. Sabarinathan, and A. Murthy. Using complete system simulation to characterize SPECjvm98 benchmarks. In *Proceedings of the 14th International Conference on Supercomputing (ICS-2000)*, pages 22–33, May 2000.
12. A. Phansalkar, A. Joshi, L. Eeckhout, and L. John. Measuring program similarity: Experiments with spec cpu benchmark suites. In *Proceedings of the 2005 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS 2005)*, pages 10–20. IEEE, 2005.
13. R. Radhakrishnan, N. Vijaykrishnan, L. K. John, A. Sivasubramaniam, J. Rubio, and J. Sabarinathan. Java runtime systems: Characterization and architectural implications. *IEEE Transactions on Computers*, 50(2):131–145, February 2001.

Application of Real-Time Object-Oriented Modeling Technique for Real-Time Computer Control

Jong-Sun Kim and Ji-Yoon Yoo

Department of Electrical Engineering, Korea University,
1, 5-ga, Anam-dong, Seongbuk-gu, Seoul, Korea
kids@korea.ac.kr

Abstract. This paper considers the design technique of the real-time control algorithm to implement the electronic interlocking system, which is the most important station control system in railway signal field. The proposed technique consists of the structure design and the detail design, which are based on the Real-time Object-Oriented Modeling (ROOM). The structure is designed by a modeling using the heuristic searching technique that catch and make out the specific requested condition. The detail design can be implemented if it may get the satisfied values through the repetitive modeling after comparing and examining the data obtained from the structure design for the more reliable and accurate system to be implemented. The technique proposed in this paper is implemented with C++ language that is easy to be transferred and compatible with the existing interfaces and the operating system is also designed and simulated by the VRTX that is a real-time operating system. This proposed technique is applied to the typical station model to prove the validity through verifying the performance of the modeling station.

1 Introduction

The recent advances in railway systems improved the efficiency of railway transportation in terms of operation speed, the number of transported units, and operation frequency. However, the increased complexity of railway systems also caused the impact of railway accidents to be more dangerous. This has driven railway system providers to focus on safety and signal preservation problems to prevent collisions and derailments that are catastrophic in nature. [1-2] The frequent operation of signals and pointers to control railway switching, arrival, stating and shunting of trains within railway stations for improved efficiency, increase the risk of accidents.[3] Therefore, for safe and efficient railway station operation, interlocking systems that provide overriding capability according to the correct sequence in case of malfunction, are being used. The most extensively used interlocking systems are relay interlocking systems and micro-processor based electronic interlocking systems.[4]

The primary design focus of relay interlocking systems is the safe operation, because system damage or breakdown does not progress into dangerous situations. Relay interlocking has been extensively used for past several decades and recognized as the most safe signal secure system that provides fast operation and prompt response. However, concerns are being raised due to standardization for interlocking

T. Srikanthan et al. (Eds.): ACSAC 2005, LNCS 3740, pp. 680 – 692, 2005.

logics, interlocking tests, design automation organization, and extension. Microcomputer based electronic interlocking devices are being developed to overcome the above-mentioned problems of relay interlocking systems and to minimize the cost and the maintenance requirements when the system needs to be rebuilt or expanded. However, it is very difficult to diagnose the root cause in case of a single device problem since there are multiple causes. Therefore, guaranteeing the stability to a level equivalent to relay interlocking systems is the main requirement for electronic interlocking systems to be benefit. This can be accomplished by careful design of both the hardware and software and their interface. The stability of the interlocking software is determined by not only its reliability and efficiency of interlocking implementation but also by the convenience of maintenance. The method of real-time system development and method of conventional data have an error of the confidence side, error detecting and error recovery, problem of exception situation processing, reusability of software of process and maintenance aspect and so on. The method for supplement shortcoming of these methods is real-time software development methodologies of object center.[6-9] These methodologies play important role in solving complexity system, maintaining and requiring problem increase of software quantity as applying to object intention concept. But, because these methods are putting emphasis on analysis than design method, which are quitting emphasis on object structure among the analysis, development of real-time software has lacking aspects. A design approach for developing an interlocking software design algorithm that improves the problems of existing systems proposed above is established in this paper. A design and modeling strategy is based on the Real-time Object-Oriented Modeling (ROOM) [9] procedure, which is the most appropriate approach in the initial stage of real-time software development, is proposed. Although it is an object-oriented method, it is a top-down design method similar to the structural analysis method based on the ROOM Technique that is effective for real-time problems; therefore, it is not only convenient for standardization, expansion, and maintenance but also can contribute to improved reliability and stability of the electronic interlocking system. The proposed technique consists of a structure design and a detailed design, which are based on ROOM. This proposed technique is applied to the typical station model in order to prove the validity through verifying the performance of the modeling station.

2 Concept of Electronic Interlocking

2.1 Electronic Interlocking System Structure

An Electronic Interlocking System is software based logical control system that uses micro-processors for communication between signal devices. It differs from existing relay interlocking systems, which are based on relays and cables for performing the same task.

An electronic interlocking system not only controls each train directly but also controls the flow of all the sections of the railroad. The Fig.1 shows the overall structure of the system. The system was consisted of EIS that process the linked logic, logic control panel (LCP) that controls the local station and track function module (TFM) that controls the field signal equipment. The main processor of the electronic

interlocking system utilizes a SUN workstation to implement a VRTX development environment. The user operates a general purpose IBM PC, its main function is display and input and output of data. The electronic interlocking system is designed to have three communication networks for the stability and low cost.

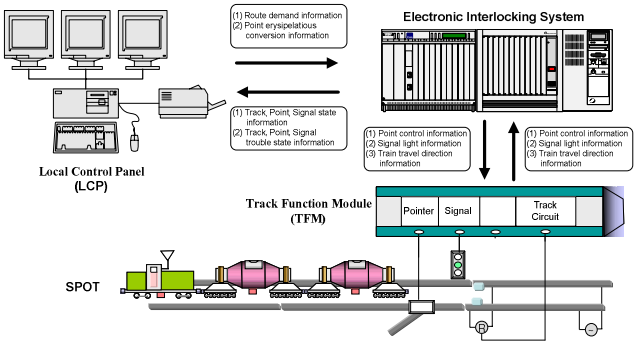


Fig. 1. Structure of an electronic interlocking system

2.2 Electronic Interlocking System Operating Strategy

The main components of a railway system are railroads and stations. There are multiple double line railways in a station for effective stopping/passing of trains and loading/unloading of cargos. When the train enters the station, the centralized traffic control (CTC) requests the permission from the interlocking system that the train can enter the station safely.

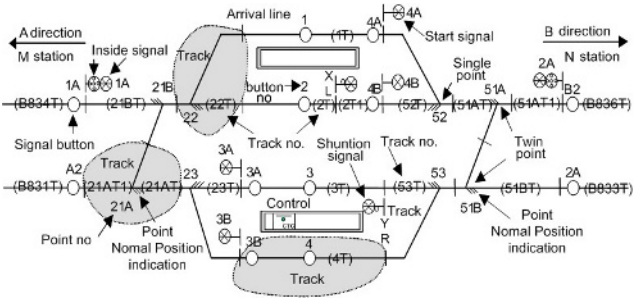


Fig. 2. Typical railway station model

When requested for permission, the electronic interlocking system checks the real-time information on the positions of the trains, and the state of the pointers and the signals in the station to determine whether it is safe to permit train entry.

A typical railway station model that shows the real-time information on the train positions, signal, and pointer is shown in Fig. 2.

3 Interlocking System Design Based on the ROOM

The design method proposed in this paper is based on the ROOM for building a precise and simple system model and for designing a recursive processor structure. The interlocking system must respond to the incoming response and process the acquired data within the limited time. The system is complicated by the fact that it must be operated in real-time. All system device components are viewed by objects and it modeling repetitively to solve the complexity of the system.

The design strategy is based on a modeling heuristic search technique (MHT), which recognizes the specific requested condition and performs the detailed designs based on this condition, as shown in Fig. 3. A message sequence chart (MSC) is formulated after analyzing the required scenario to model the internal system structure. The objects that are recognized during the internal structure modeling are modeled to determine their logical relations. After such system modeling, the optimized model is created by the required scenario.

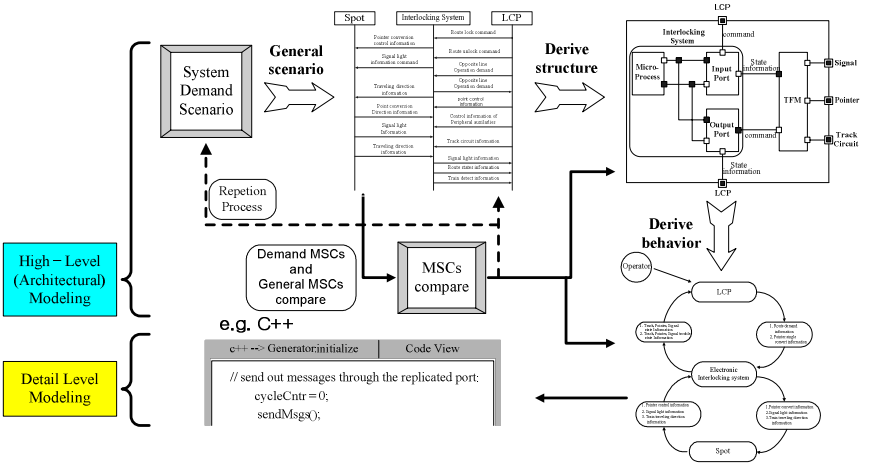


Fig. 3. Software Design Strategy for the Interlocking System

A gradual and repetitive approach for modeling is created during this process, and each modeling period processes the increments of requested conditions. Repetition occurs when the classes created from the previous modeling period is re-examined.
As we can see the Fig. 4, the modeling strategy 1) utilizes the advantages of the new paradigm (object orientation), 2) includes the powerful concept of real-time, and 3) makes it easy to build a precise and simple system model. In addition, it is possible to recognize the elucidative system structure and records and the requirements and design flaws can detected in an early stage since an active model is provided by surmising all concepts of leveling.

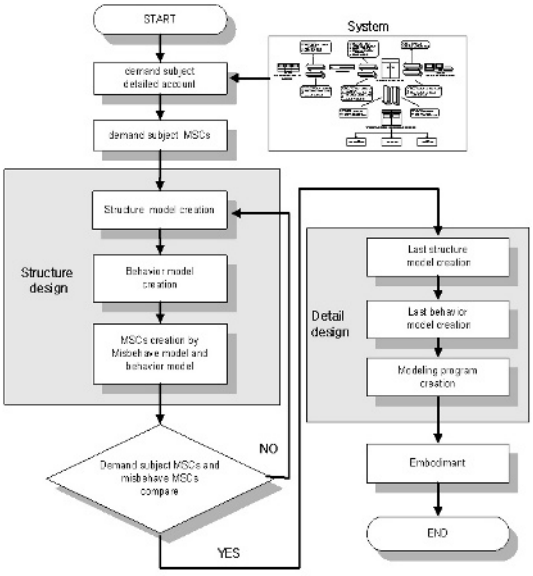


Fig. 4. Flow chart of real-time control software architectures modeling strategy for Electronic Interlocking System

3.1 Scenario for System Requirements

Scenario is a basic method for describing the system requirements. Scenarios identify the usage of the entire system as well as the sequence for the inner elements to follow. Because scenarios focus on the objects and the sequence of the messages that the object follow, they are very useful for deduce their elements and requirements. The scenarios of the requirements for the electronic interlocking system are shown in Scenario 1), 2), 3).

Scenario 1) Route Setup

#1. LCP sends route setup command to Electronic Interlocking System.
#2. Electronic Interlocking System demands the state information of the track.
#3. Electronic Interlocking System receives the state information of track and sends conversion command to the corresponding pointers.
#4. Receive the state information of the track again, after conversion of the Pointer.
#5. Sends progress signal light command.
#6. Receive the state information of the track again, after progress signal light.
#7. Transmit the results to LCP, after implementation of all route setups.

Scenario 2) Route-Cancellation

#1. LCP sends route cancel command to EIS.
#2. Electronic Interlocking System demands the state information of the track.
#3. Electronic Interlocking System receives the state information of the track and sends stop signal light command to the corresponding signal.

#4. Receive the state information of the track, after implementation of the stop signal light command.
#5. EIS sends unlock command to the corresponding pointers.
#6. Receives the state information of the track again, after implementation of unlock.
#7. Transmit the result to LCP, after implementation of all route cancellations.

Scenario 3) Single Conversion of Pointer

#1. LCP sends pointer single conversion command.
#2. Electronic Interlocking System demands the state information of the track.
#3. Electronic Interlocking System demands the lock state to the corresponding pointer, after identification of the state information of the track.
#4. Demand the direction state information of the pointer, after identification of the lock state information of the pointer.
#5. Convert the direction of the pointer, after identification of the direction state information of the pointer.
#6. Transmits the result to LCP when termination of direction conversion of the pointer.

3.2 Interlocking System Architecture Modeling

Fig. 5 shows the initial model of high-level system limits. In this fig. 5, the system is viewed as a block box, and it is shown how the system outer interface is correlated with outer objects (LCP and field installation).

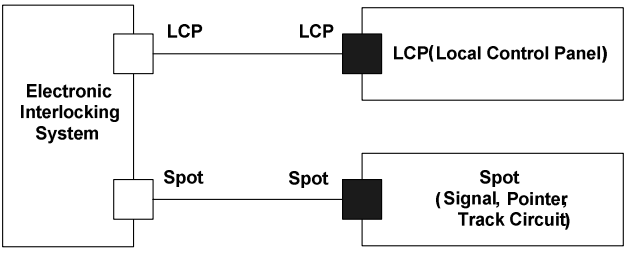


Fig. 5. Initial model : High-Level System Limits

The MSC can be created as shown in Fig. 6 according to the initial model of system limits. The system is viewed as a black box from the viewpoint of the electronic interlocking system, LCP, and dissolved signal installations (pointer, signal, and track circuits). Because initial modeling is based on objects sampled from an overall system glance, the details of the object characteristics do not appear clearly and the correlation between objects is not well defined.

The stage that follows that of the initial model is shown in Fig. 7, where the system is viewed from the outer and inner pointer of view. This is the detailed view of the system shown in Fig. 5, where the MSC is reconstructed by the system structure

model. At this stage, system is not a black box as shown in Fig. 5 and system is embodied by the LCP, electronic interlocking system, signals, trains, track circuits and so on. The system requirement scenario is represented as the MSC.

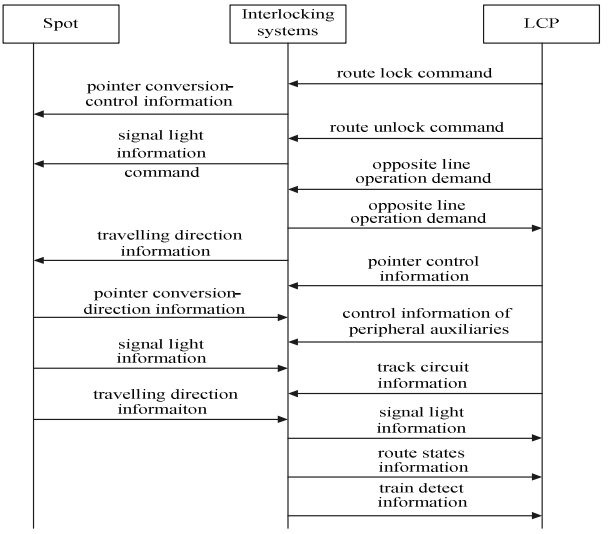


Fig. 6. Initial Limits of the MSC

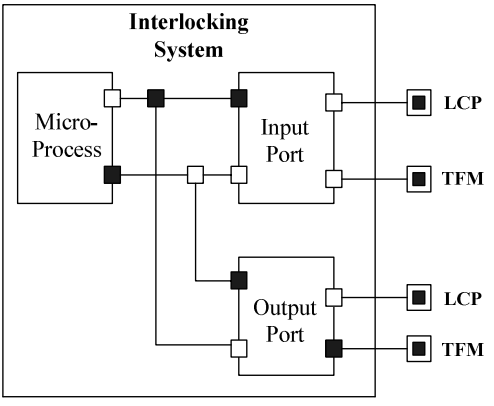


Fig. 7. High-Level System Structure

The behavior between the user and system is shown in Fig. 8, where rounded rectangles represent states or substrates. The various and complicated last scenario can be defined recurrently by the other scenarios. Since the request condition scenario focuses on the objects and message sequences between objects, they are particularly useful in deducing the requirements of their elements. The following step presents the

interlocking system in Fig. 8 in more detail. The behavior and interface of the entire system are not clear yet and the systematic relation between users is not clear as well. Fig. 9 shows all the inner devices of interlocking system and the systematic message transfer routines.

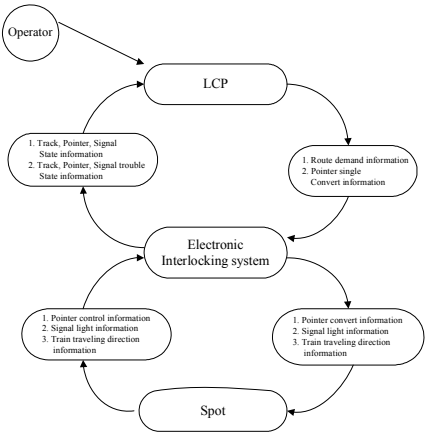


Fig. 8. User Behavior

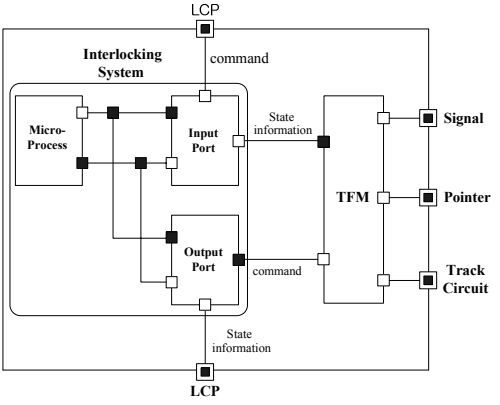


Fig. 9. The Last high-level System Structure

The final MSC created as a result a repetitive process for each object's message interface relations is shown in detail in Fig. 10. It represents the detailed contents of the message communication between each object. The errors or abnormal states can be included by repetitive process during the maintenance.

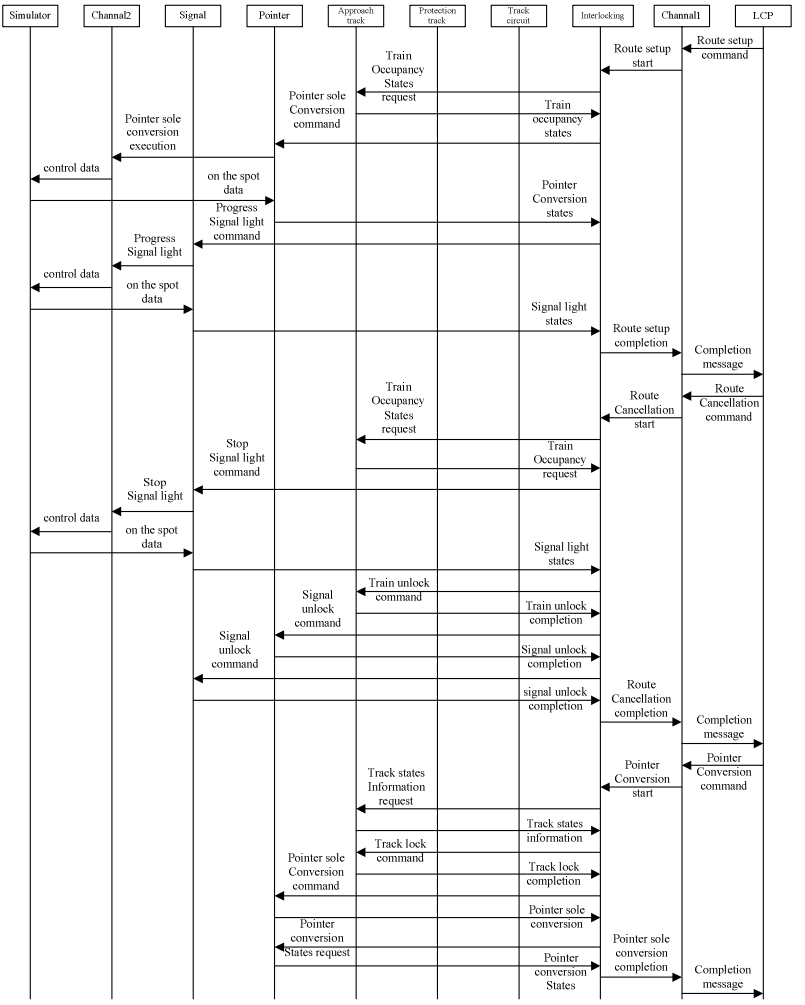


Fig. 10. The Last Interlocking Scenario MSCs

4 Simulation Result

The proposed software was tested on a model of a railway station of the city railway system to verify that the precise chain relations between each element and controls are as predicted. The reliability and the efficiency of the development design strategy and the proposed control algorithm are also verified. Since the proposed method cannot be tested on an actual railway system, a railway simulator was used. Fig. 11 shows the configuration of the simulation experiment, which plays the role of the LCP and the on-the-spot simulator.

The Simulator saves the status information of on-the-spot signals, pointers, and tracks, outputs on-the-spot data when the interlocking system sends a request. It

behaves exactly like a real railway system that the status information of signals or pointers is transferred on the request. The model of the railway station shown in Fig. 2, used in the simulation, consists of a total of eight regulation routes, where there are 4 routes each in opposite directions (up-line and down-line). Based on the requirement scenario of the interlocking system for each regulation route, verifications were made for various scenarios that originate in lock, detector locking, signal control and approach locking region-mutual chain relations. The high efficiency and reliability are verified by testing whether all approach locking regions succeed in locking precisely and by testing route-cancellation-setup, sole point conversion, etc. In circumstances identical to a real spot. It is verified that the interlocking function of the algorithm designed based on the proposed technique can monitor route locking, lock pointers, detector, signal control, block locking, direction lever, particle lever, chasing vehicles, etc. in a short period of time.

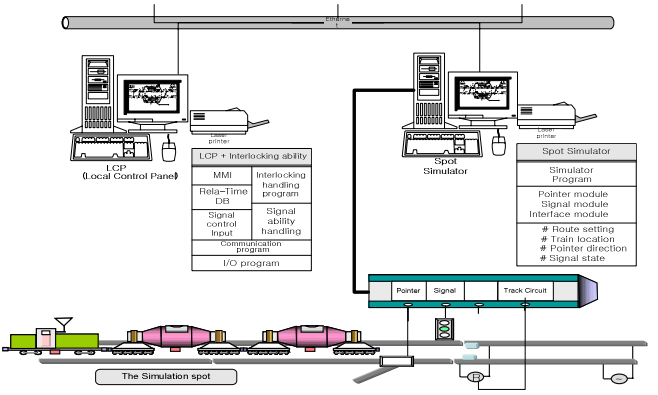


Fig. 11. Configuration of Simulation Equipment

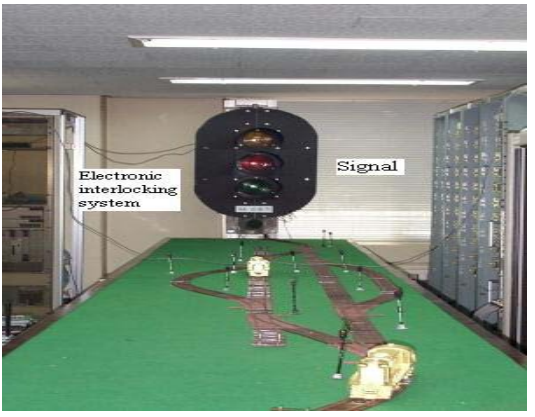


Fig. 12. Experimental setup

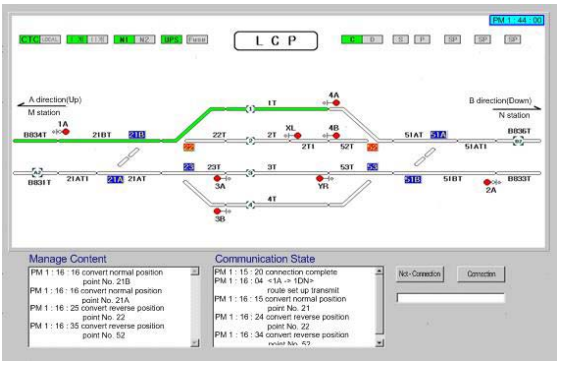


Fig. 13. LCP tool

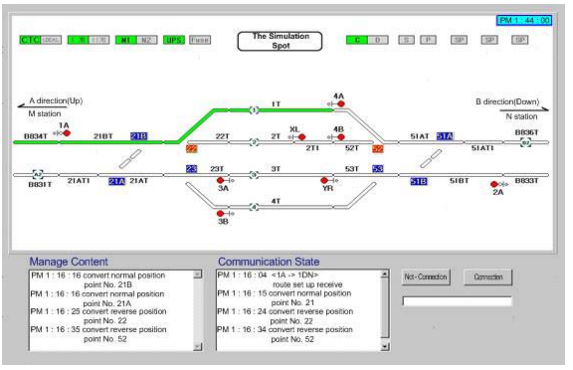


Fig. 14. Spot simulator tool

In addition, it was also verified that occurrences of faults during separate conversions of each signal and point in case of frequently operation, and intentionally inserted errors when creating control-data files could be detected. The experimental setup is shown in Fig. 12, 13 and Fig. 14 show the man-machine interface (MMI) of experiment equipments. For a regulation route {1A, 1} working under computer simulation, the exact operating status, as shown in the following, was verified.

(1) Reverse lever of inner signals (1A)
(2) Check whether opposite route is set to the down direction of No. 1 line.
(3) Convert the pointer No. 22 to the reverse position and convert the pointer No. 21B to the normal (N) position.
(4) Check whether the route (B834T, 21BT, 22T, 1T) is set up to the down direction of No. 1 line.
(5) Lock pointer No. 21, No. 22
(6) Verify whether there is another train is in the rail.
(7) Display the yellow light in the grounds signal (1A).
(8) Re-position the reversed lever to the normal position (N) when the train is entering the station.

(9) Lock the corresponding pointers until the train passes the relational track circuit.
(10) Unlock the pointers of track circuits whenever the train passes each track circuit.
(11) Unlock the opposite route when the train reaches the arrival line.

Table 1. Simulation Results for Regulation Route

Verif Cont / Reg. route	Pointer state	lock-ing	Detector locking	Signal control	Route locking	Approach locking	Train occupancy track section	Signal state	Approach locking timer	Result
{1A,1}	21:N 22:R 52:R 51:N	21 22 52 51	21BT 22T 1T	21BT 22T 1T	(21BT) (22T)	B834T	B834T 21BT 22T 1T	G R R	90sec	Success
{4A,B2}	52:R 51:N	52 51	52T 51AT 51AT 1B836T	52T 51AT 51AT1 B836T	52T 51AT 51AT1	1T	1T 52T 52T,51AT B836T	G R R Y	30sec	Success
{1A,2}	21:N 22:N 52:N 51:N	21 22 52 51	21BT 22T 2T 2T1	21BT 22T 2T 2T1	(21BT) (22T) 2T1	B834T	B83T 22T,2T 2T	G R R	90sec	Success
{4B,B2}	52:N 51:N	52 51	52T 51AT 51AT1 B836T	52T 51AT 51AT1 B836T	52T 51AT 51AT1	2T	2T 52T 52T,51AT1 B836T	G R R Y	30sec	Success
{2A,3}	51:N 53:N 23:N 21:N	51 53 23 21	51BT 53T 3T	51BT 53T 3T	(51BT) (53T)	B833T	B833T 53T,3T 3T	G R R	90sec	Success
{3A,A2}	23:N 21:N	23 21	23T 21AT 21AT1 B831T	23T 21AT 21AT1 B831T	23T 21AT 21AT1	3T	3T 23T 23T,21AT B831T	G R R Y	30sec	Success
{2A,4}	51:N 53:R 23:R 21:N	51 53 23 21	51BT 53T 4T	51BT 53T 4T	(51BT) (53T)	B833T	B833T 53T,4T 4T	G R R	90sec	Success
{3B,A2}	23:R 21:N	23 21	23T 21AT 21AT1 B831T	23T 21AT 21AT1 B831	23T 21AT 21AT1	4T	4T 23T 23T,21AT B831T	G R R Y	30sec	Success

Pointer states are converted from no. 21 and 51 to the normal position (N), and from no. 22 and 52 to the reverse position (R) and locked. In addition, it was locked so that the point is not converted the by trains and vehicles in case there are trains or vehicles on the track circuit. Track circuit 21BT, 22T, and 1T controlled the signal lights, and in direction locking, trains or vehicles go into the direction locking, track 21BT, 22T were locked. So as not to convert point on the route by trains or vehicles until it passes relational track circuits though repositions the signal lever. In an approach locking, signal is locked so as not to change route (B834T) unconditionally route until 90 seconds goes on after lighting stop sign in signal. Verified results of the regulation route are shown in Table 1. As can be seen from the performance verification results that the chain relations between signal equipments are correct for each regulation route. The results showed that in case an error occurs in each signal or pointer, the correct route was not transmitted. The fact that the data automatically generated by the IIKBAG[5] provided accurate results for the LCP transmitted route-command verified the reliable compatibility.

5 Conclusion

A new reliable on-line interlocking handle control algorithm providing stability as well as standardization, expansion ability, and convenience of maintenance has been proposed. The new design strategy was designed so that it provides a reliable control system through repetitive process modeling. Another design criterion was to be able to verify the control system requirements by modeling systems during a short period, which enabled detection of design flaws and thus enhanced the precision.

The new method designs the control algorithm as a module for each unit, and the complex data structure of the interlocking data was easily recognizable since it was designed as a file structure that displays interlocking conditions similar to the connection status of a railway line. The performance of the proposed algorithm was verified using a representative model station and the validity and the efficiency was verified by results that showed accurate interlocking relations and execution for all scenarios.

References

1. H. Yoshimura, S. Yoshikoshi : Railway Signal. JASI Toyko (1983) 9 – 21
2. E.J. Phillips Jr, : Railroad Operation and Railway Signaling. Simmons- Boardman Publishing, N.Y. (1953)
3. A.H. Cribbens, : Solid-State Interlocking (SSI) : An Integrated Electronic Signaling System For Mainline Railway. IEE Proc. Vol. 134. MAY (1987) 148 – 158
4. C.R. Brown, R.D. Hollands, D. Barton, : Continuous automatic train control and safety system using microprocessors. Proc. Int'1 Conf. Electric Railway Systems for a New Centry, London UK (1987)
5. Y.S. Ko, J.S. Kim, : Software Development for Auto-Generation of Interlocking Knowledgebase Using Artificial Intelligence Approach. KIEE Trans, Vol. 48A, No 6, June (1999) 800 – 806
6. I. Jacobson. : Object-Oriented Software Engineering. Addison Wesley. (1992).
7. J.Rumbaugh, M.Blanha, W.Premerlani, F.Eddy, W.Lorensen. : Object-Oriented Modeling and Design. Prentice Hall. (1991)
8. E. Gamma, R. Helm, R. Johnson, and J.Vlissides. : Design Patterns : Elements of Reusable Object-Oriencted Software, Reading. MA: Addison-Wesley. (1994)
9. B. Selic, G. Gullekson, and P.T.Ward. : Real-Time Object-Oriented Modeling. John Wiley & Sons. (1994)

VLSI Performance Evaluation and Analysis of Systolic and Semisystolic Finite Field Multipliers

Ravi Kumar Satzoda and Chip-Hong Chang

Centre for High Performance Embedded Systems,
Nanyang Technological University, Singapore, 639798
{rksatzoda, echchang}@ntu.edu.sg

Abstract. Finite field multiplication in $GF(2^m)$ is an ineluctable operation in elliptic curve cryptography. The objective of this paper is to survey fast and efficient hardware implementations of systolic and semisystolic finite field multipliers in $GF(2^m)$ with two algorithmic schemes – LSB-first and MSB-first. These algorithms have been mapped to seven variants of recently proposed array-type finite-field multiplier implementations with different input-output configurations. The relative VLSI performance merits of these ASIC prototypes with respect to their field orders are evaluated and compared under uniform constraints and in properly defined simulation runs on a Synopsys environment using the TSMC 0.18μm CMOS standard cell library. The results of the simulation provide an insight into the behavior of various configurations of array-type finite-field multiplier so that system architect can use them to determine the most appropriate finite field multiplier topology for required design features.

1 Introduction

In recent years, there has been a resurgence of interest in the pursuit of VLSI efficient finite field arithmetic due mainly to its application in public key cryptography. One strong merit of public key cryptography in information security lies in its ease of key management [1]. RSA, Deffie-Hellman and Elliptic Curve Cryptography (ECC) are few proven and prevailingly used public key encryption algorithms reported in the literature [1, 2]. ECC has been an active field of research because ECC provides higher level of security at smaller key lengths than RSA [4]. This helps to reduce the cost of implementation both in software and hardware. However, software implementations of ECC are comparatively slower. Therefore, it is desired to design efficient architectures in hardware for cryptographic algorithms.

All the operations in ECC are done in finite fields $GF(p^m)$. When $p = 2$, elements in Galois field $GF(2^m)$ are represented as binary numbers and all the operations can be performed using standard logic gates in binary domain. Addition in $GF(2^m)$ is simpler than normal binary addition as the sum can be obtained by bit-wise XOR operation of the two operands and the latency is independent of the operand sizes since there is no carry chain [16]. However, multiplication is more complicated because it involves calculating the modulus with

T. Srikanthan et al. (Eds.): ACSAC 2005, LNCS 3740, pp. 693–706, 2005.

respect to an irreducible polynomial $f(x)$, i.e. $a(x) \times b(x)$ mod $f(x)$ if the field elements are expressed in polynomial bases. Important techniques for software and hardware implementations of finite field multiplication have been reported in literature [3,4,5,6,7,8,9,10,11,12,13,14,15], [18], [20]. Several software implementations of finite field multiplication are listed in [3] whereas efficient hardware architectures are described separately in [4,5,6,7,8,9,10,11,12,13,14,15], [18], [20]. There are generally three different categories into which reported multipliers can be sieved. [4] illustrates a multiplier of the first category where reduction is given higher priority than multiplication. Multiplication is performed by simple *AND-XOR* network. However, reduction is more complicated and efficient implementations for reduction are designed. In [5,6,7,8,9,20], a variety of implementations of Mastrovito, Karatsuba and Massey-Omura multipliers have been reported, which form the second category. The third category comprises systolic and semisystolic multipliers which perform multiplication and reduction simultaneously.

Systolic and semisystolic finite field multipliers are reported separately in [10, 11, 12, 13, 14, 15]. In this paper, we survey some recently reported systolic and semisystolic finite field multipliers. We describe these architectures briefly followed by important results reported in [11, 12, 13, 14]. Different input-output configurations like bit-serial, bit-parallel and digit-serial are discussed. In contrast to performance results reported for these multipliers, which are determined by analytical models of area-time complexity, the VLSI performance metrics of the reported designs are realistically evaluated for different field sizes by mapping the various architectures to ASICs using the 0.18μm CMOS technology libraries. Based on the results of our simulation, a system architect can decide on a suitable systolic or semisystolic array-type finite field multiplier for designing an optimized ECC cryptosystem with the desired characteristics of chip area, throughput rate and power consumption.

This paper is structured as follows. Section 2 summarizes multiplication in $GF(2^m)$ followed by the description of various systolic and semisystolic architectures we intend to synthesize and compare in Section 3. In Section 4, simulation and synthesis results of ASIC implementations are analyzed and discussed in detail. We conclude and discuss future research in Section 5.

2 Multiplication in $GF(2^m)$

In this section we describe multiplication in Galois field $GF(2^m)$. More details on finite field arithmetic can be found in [16], [19]. Finite field $GF(2^m)$ contains 2^m elements which is an extension of $GF(2)$ where the elements $\in \{0, 1\}$. Elements in $GF(2^m)$ can be expressed in two different bases – polynomial and normal bases [16]. All the architectures described in this paper are based on polynomial basis representation. Each element in $GF(2^m)$ is represented as a binary polynomial of degree less than m. If $a \in GF(2^m)$, then a is represented as

$$a(x) = a_{m-1}x^{m-1} + a_{m-2}x^{m-2} + \cdots + a_1x + a_0 \quad (1)$$

where $a_i \in \{0,1\}, 0 \leq i \leq m-1$. Thus a field element, $a(x)$ can be denoted by binary string $(a_{m-1}a_{m-2}\ldots a_1a_0)$.

Let $a(x) = (a_{m-1}a_{m-2}\ldots a_1a_0)$ and $b(x) = (b_{m-1}b_{m-2}\ldots b_1b_0)$ be two elements in $GF(2^m)$. Multiplication of $a(x)$ and $b(x)$ in $GF(2^m)$ involves an irreducible polynomial $f(x)$ given by

$$f(x) = x^m + f_{m-1}x^{m-1} + f_{m-2}x^{m-2} + \cdots + f_1x + f_0 = x^m + \sum_{i=0}^{m-1} f_ix^i \quad (2)$$

where $f_i \in \{0,1\}$. $f(x)$ is called reduction polynomial. The product of $a(x)$ and $b(x)$ is defined as

$$c(x) = \sum_{i=0}^{m-1} c_ix^i = a(x) \times b(x) \bmod f(x) = (\sum_{i=0}^{m-1} a_ix^i)(\sum_{i=0}^{m-1} b_ix^i) \bmod f(x)\ . \quad (3)$$

In (3), $a(x) \times b(x)$ gives $c'(x) = \sum_{i=0}^{2m-2} c_i'x^i$, a polynomial of degree $2m-2$ or $2m-1$ bit binary string. $c'(x)$ is reduced by $f(x)$ to give $c(x)$ which is a polynomial of degree $m-1$. $c(x)$ is the remainder obtained on dividing $c'(x)$ by $f(x)$. Since there is no carry chain for addition in $GF(2^m)$, multiplying two m-bit vectors produces a $2m-1$ bit product, as opposed to the $2m$ bit product in normal binary multiplication. Multiplication is implemented in hardware/software in two different ways. The first method determines $c'(x) = a(x) \times b(x)$ and then uses the reduction polynomial $f(x)$ to reduce $c'(x)$ to get $c(x) = c'(x) \bmod f(x)$. In the second method, the partial-product bits obtained by multiplying $a(x)$ by b_i are reduced immediately by $f(x)$.

The systolic and semisystolic multipliers are illustrated based on the second method. All these multipliers are implemented based on the following two schemes.

LSB-first method:
$c(x) = a(x)b(x) \bmod f(x)$
$= b_0a(x) + b_1[a(x)x \bmod f(x)] + b_2[a(x)x^2 \bmod f(x)] + \cdots + b_{m-1}[a(x)x^{m-1} \bmod f(x)]$
MSB-first method:
$c(x) = a(x)b(x) \bmod f(x)$
$= \{\cdots[a(x)b_{m-1}x \bmod f(x) + a(x)b_{m-2}]x \bmod f(x) + \cdots + a(x)b_1\}x \bmod f(x) + a(x)b_0$

The two methods differ in the order the bits of the multiplier $b(x)$ are accessed. In LSB-first method, the multiplication starts with the LSB of $b(x)$ whereas the latter starts with MSB of $b(x)$. The pseudo-codes for their implementations are shown in Algorithm 1 [15] and Algorithm 2 [14].

`Algorithm 1: LSB-first`
`Input: a(x),b(x),f(x)`
`Output: c(x) = a(x)b(x) mod f(x)`
$t_j^{(0)} = 0,\ for\ 0 \leq j \leq m-1$
$a_{-1}^{(i)} = 0,\ for\ 0 \leq i \leq m$
$a_j^{(0)} = 0,\ for\ 0 \leq j \leq m-1$
$for\ i\ =\ 1\ to\ m$
$\{for\ j\ =\ m-1\ to\ 0$
$\{a_j^{(i)} = a_{j-1}^{(i-1)} + a_{m-1}^{(i-1)} f_j$
$t_j^{(i)} = a_j^{(i-1)} b_{i-1} + t_j^{(i-1)}\}\}$
$c(x) = t^{(m)}(x)$

`Algorithm 2: MSB-first`
`Input: a(x),b(x),f(x)`
`Output: c(x) = a(x)b(x) mod f(x)`
$t_j^{(0)} = 0,\ for\ 0 \leq j \leq m-1$
$t_0^{(i)} = 0,\ for\ 1 \leq i \leq m$
$for\ i\ =\ 1\ to\ m$
$\{for\ j\ =\ m-1\ to\ 0$
$\{t_j^{(i)} = t_{m-1}^{(j-1)} f_j + b_{m-i} a_j + t_{j-1}^{(i-1)}\}\}$
$c(x) = t^{(m)}(x)$

Both algorithms stated above have been realized with bit-serial and bit-parallel architectures in [10, 11, 12, 13, 14, 15] [18] [20]. Bit-parallel multipliers provide higher throughput at a cost of increased hardware. Hardware complexity is an important criterion for public key cryptography on smart card in view of the finite field operations with large word length. Therefore, bit-serial multipliers are preferred over bit-parallel counterparts in those applications. However, latency becomes an issue in bit-serial multipliers. Digit-serial multipliers have also been reported to reduce latency but digitizing the design increases hardware complexity and combinational delay as compared to bit-serial multipliers. In this paper, we survey important systolic and semisystolic architectures that have been reported in the last ten years. A comprehensive analysis of the ASIC prototypes of the seven different architectures is provided.

3 Architectures of Finite Field Multipliers

In this section, we describe some systolic and semisystolic multiplier architectures. A quantitative analysis in terms of gate count and latency is presented here. Algorithm to architecture mapping is described briefly for each of the multipliers to be evaluated.

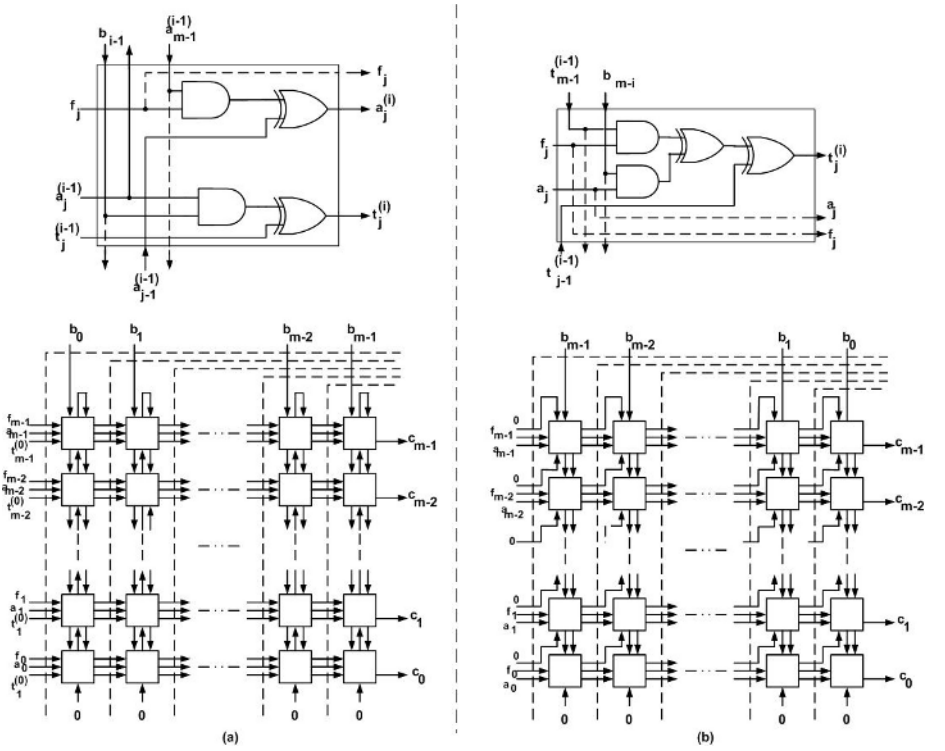


Fig. 1. (a) Basic cell and architecture for LSB-first bit-parallel finite field multiplier (b) Basic cell and architecture for MSB-first bit-parallel finite field multiplier

3.1 Bit-Serial/Parallel Array Multipliers

Bit-parallel Pipelined LSB-First Semisystolic Multiplier. In [11], a bit-level pipelined parallel-in parallel-out LSB-first semisystolic multiplier is proposed based on `Algorithm 1`. The basic cell, which computes $a_j^{(i)}$ and $t_j^{(i)}$ at `Step i` is shown in Fig. 1(a) with the architecture of an m-bit multiplier. The dotted lines indicate cutsets where latches are placed to pipeline it. This implementation comprises m^2 basic cells and $3m^2$ l-bit latches. The latency of this implementation is $m+1$ clock cycles. From Fig. 1(a), we see that the critical path consists of one two-input AND gate followed by one two-input XOR gate, i.e. $T_{2-AND} + T_{2-XOR}$.

Bit-parallel Pipelined MSB-First Semisystolic Multiplier. Based on `Algorithm 2`, a bit-level pipelined MSB-first semisystolic multiplier is implemented as shown in Fig. 1(b) which was reported in [11, 18]. The basic cell in Fig. 1(b) computes $a_j^{(i)}$ and $t_j^{(i)}$ at `Step i`. Unlike the LSB-first scheme, $a(x)$ is multiplied by b_{m-1} first instead of b_0. Moreover, $a(x)$ in $(i-1)$-th iteration ($a_j^{(i-1)}$ bits) need not be latched according to the algorithm. The critical path is limited by the delay

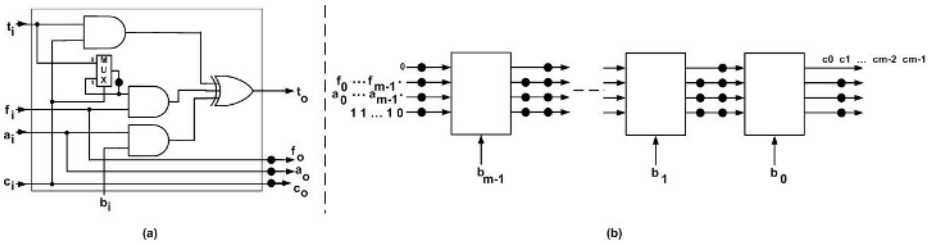


Fig. 2. (a) Basic cell and (b) Bit-serial MSB-first finite field multiplier

in the basic cell and is given by $T_{2-AND} + 2T_{2-XOR}$. Hence the delay is longer than LSB-first multiplier by T_{2-XOR}. Moreover, the number of cycles to complete the operation remains the same, i.e., $m + 1$ clock cycles. In addition, the number of 1-bit latches is reduced to m^2 in MSB-first implementation because only $t_j^{(i)}$s have to be latched in each basic cell.

Serial-in Serial-out MSB-First Multiplier. In [12], a serial-in serial-out multiplier is described. It is also based on the MSB-first algorithm. The basic cell for this architecture is shown in Fig. 2(a). It is a modified version of the basic cell shown in Fig. 1(b), which was used in the MSB-first parallel architecture. The two XOR gates in Fig. 1(b) are replaced with a 3-input XOR gate. There is an additional multiplexer MUX and an AND gate. The extra circuitry enables the signals to propagate in a serial-in serial-out fashion. It should be noted that the subscript i for t_i, f_i, a_i and c_i in Fig. 2(a), indicates the input bit position rather than the iteration number. • represents a delay element. c_i is the control signal to select the inputs of the multiplexer. The serial-input is controlled by a control sequence $0111\cdots11$ with length m. The first bit, *0* of the control sequence enters the array one cycle ahead of the MSB of $a(x)$. This *0* control bit selects the MSB of $t(x)$ of the $(i-1)$-th iteration to be ANDed with f_j. Thereafter, when the control signal sets to *1*, the MSB of $t(x)$ of the $(i-1)$-th iteration is latched back into the multiplexer as shown in Fig. 2(a). In this way, the equation, $t_j^{(i)} = t_{m-1}^{(i-1)} f_j + b_{m-i} a_j + t_{j-1}^{(i-1)}$ in `Algorithm 2` for the calculation of $t_j^{(i)}$ (i.e., the intermediate product bit j at the i-th iteration) is realized for a serial-input.

Fig. 2(b) shows the serial-in serial-out architecture of an m-bit multiplier in $GF(2^m)$. It comprises m basic cells and has a latency of $3m$ clock cycles with a throughput rate of $1/m$. The combinational delay is equivalent to $T_{2-AND} + 2T_{2-XOR}$ (a 3-input XOR gate is considered as equivalent to two 2-input XOR gates). The logic complexity of the basic cell is given by three 2-input AND gates, one 3-input XOR gate, nine 1-bit latches and one switch (multiplexer).

3.2 Digit Serial Systolic Multipliers

Two digit-serial systolic multipliers are described in [13] and [14]. Both of them are based on the MSB-first algorithm. Here we highlight some of the salient

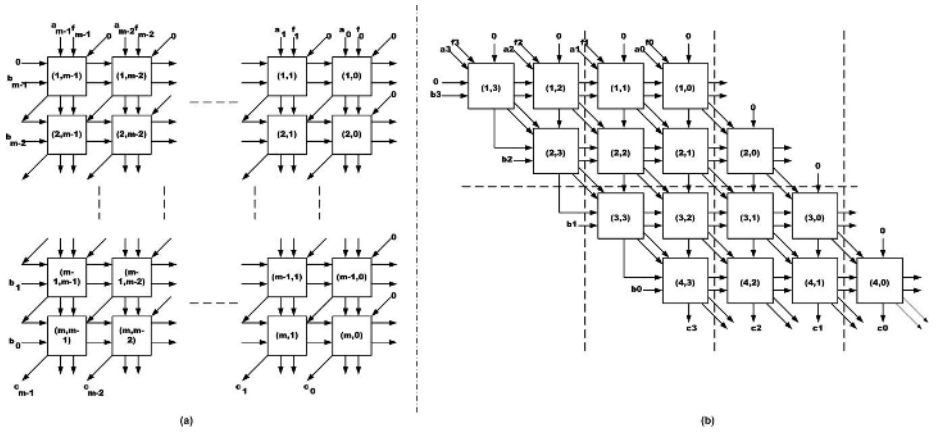


Fig. 3. (a) DG as in [13] (b) DG as in [14]

features of these architectures. The dependency graph (DG) shown in Fig. 3(a) is modified in [13] and [14] to create a digit-serial multipliers. The problem encountered in both architectures is how to project the DG in the east direction to obtain a one-dimensional signal flow graph. In [13], for a digit length L, a basic module comprising L^2 cells is selected. These cells form a square grid of L cells in the `x` and `y` directions. Extra circuitry consisting of 4-input XOR gates is used in the basic module to overcome the bidirectional signal flow in the DG of Fig. 3(a). The derivation of the architecture and the basic module are detailed in [13]. For a digit size of L, each basic cell has a logic complexity of $L-1$ 2-input XOR gates, $2L^2+L$ 2-input AND gates, $L-1$ 4-input XOR gates, $10L$ 1-bit latches and $2L$ switches (2-to-1 multiplexers) [13].

In [14], DG is modified to prevent the use of 4-input XOR gates. The authors transform indices of the DG to form a new DG where each row in DG (see Fig. 3(a)) is shifted towards the right by one basic cell, i.e., Cell $(2, m-1)$ is placed under Cell $(1, m-2)$ instead of Cell $(1, m-1)$. The new DG is then divided into m/L parts horizontally where each part comprises L rows. Fig. 3(b) shows the DG partitioning for a 4-bit multiplier where $m=4$ and $L=2$. Each part is further divided into $m/L+1$ regions vertically. Due to the transformation of indices, the regions have different number of cells. Each basic module as shown in [14] has L^2 cells. In contrast, the cutsets in Fig. 3(b) do not form equal regions. Latches and multiplexers are introduced between cells in the basic module to accommodate the cutsets. More information pertaining to the basic module and cutsets can be obtained from [14]. The complexity of the basic module is given by $2L^2$ 2-input AND gates, L^2 3-input XOR gates, $8L+2$ 1-bit latches and $2L$ switches.

3.3 Generalized Cellular-Array Multipliers

In [11], two generalized cellular-array multipliers are proposed based on the LSB-first and MSB-first algorithms presented earlier. The prefix 'generalized' refers

f_i	y_{in}	h_i	y_{out}
0	0	0	0
0	1	0	1
1	0	1	1
1	1	0	1

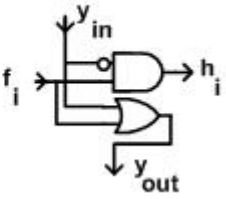


Fig. 4. Truth table to compute H vector and LCELL

to the applicability of the structure to varying field order, i.e., the multiplier is designed for multiplication over $GF(2^M)$ but the primitive polynomial $f(x)$ is of order m such that $m \leq M$. The higher order bits of primitive polynomial are padded with zeros.

The authors of [11] first determine the order of the primitive polynomial by bit-locator cells – LCELL. The LCELL is derived from the truth table of Fig. 4 as given in [11]. The H vector from LCELLs is an M-bit binary string with only one non-zero bit. The only *1* bit corresponds to the field order. For example, if $p(x) = x^5 + x + 1$ and $M = 9$, then $H = 00010000$. This is followed by modifying both algorithms to incorporate programmability of field order m. The array is now made up of MCELLs (multiplier cells), which are based on the modified algorithms in [11]. Based on the architectures presented in [11], the complexity of the basic cell in the generalized LSB-first array multiplier is increased by four gates compared to the fixed order LSB-first multiplier described in Sec. 3.1. Similarly the complexity is increased by 3 gates in MSB-first programmable multiplier compared to its fixed order counterpart. LCELLs form extra circuitries that are needed for the precomputation for the generalized architectures. One disadvantage of generalized architectures is that the critical path is longer. It runs vertically in the array due to signals propagated vertically by y_{out} in the array. As a result, the clocking frequency decreases. The critical path is equivalent to $(m-1)T_{2-OR}$ in both cases.

4 ASIC Implementation Results

In this section, we present simulation results of the architectures discussed in Sec. 3. These simulation results give a better estimate of VLSI performance metrics – silicon area, critical path delay and dynamic power dissipation. Structural VHDL codes were generated automatically using C programs to facilitate more rigorous simulations of different architectures under varying field size. The designs were synthesized, optimized and simulated using Synopsys Design Compiler version 2004.06. The finite field multipliers were optimized with consistent constraints set in Design Compiler. The input and output loads were set to 0.8pF and 0.9pF respectively. Cross boundary optimization was enabled to harness further area

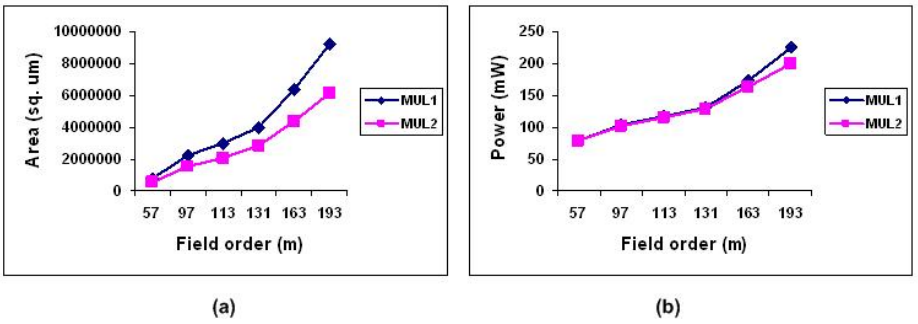


Fig. 5. (a) Area results and (b) Power results of MUL1 and MUL2

reduction. Power simulations were performed using Synopsys Power Compiler with a set of thousand random input vectors at a clock frequency of 20MHz. The area results are also obtained at this clock frequency. However, the circuits can meet this timing constraint comfortably. Thus, they can be further optimized for faster clocks at the expense of higher area. Nine different derivatives of finite field multipliers from Sec. 3 were simulated.

MUL1 and MUL2 represent the parallel-in parallel-out LSB-first and MSB-first multipliers respectively from [11]. The bit-serial architecture in [12] is named MUL3. Two additional bit-serial multipliers MUL4_BS and MUL5_BS, were obtained from the digit serial multipliers stated in [13] and [14] by setting the digit size to 1. MUL1, MUL2, MUL3, MUL4_BS, MUL5_BS, MUL4_DS and MUL5_DS were developed for different field orders 113, 131, 163 and 193 as specified by SEC2 [17]. The strength of ECC with a key length of 193 bits is equivalent to that of a 1536-bit key length in RSA [17]. The generalized bit-parallel architectures reported in [11] – MUL6 and MUL7, were designed for $M = 193$ as that is the maximum order we are interested in this paper. In addition to the above field orders, the first seven multipliers for lower field orders - 57 and 97 were also implemented.

Bit-level parallel-in parallel-out implementations [11]: MUL1 and MUL2 fall in this category. Fig. 5(a) and (b) show the comparison of the silicon area and dynamic power dissipation of MUL1 and MUL2, respectively. From Fig. 5(a), we see that MUL1 (LSB-first) occupies higher silicon area than MUL2 (MSB-first). The difference is attributed mainly to the extra latches in MUL2. It was shown previously in Sec. 3 that MUL2 employs m^2 latches whereas MUL1 has $3m^2$ latches. However, there is a marked increase in the difference between the two designs as m increases. As m increases towards 193, the area of MUL2 approaches that of MUL1 with a lower field order. From Fig. 5(b), we see that both MUL1 and MUL2 dissipate around the same amount of dynamic power with MUL2 having a slight edge over MUL1. This shows the ascendancy of MSB-first algorithm in terms of silicon area and dynamic power dissipation, particularly for higher field orders. The critical path delays of both designs are comparable and is approximately 1.36 ns.

Serial-in serial-out implementations [12, 13, 14]: MUL3 in [12] is a bit serial multiplier whereas those reported in [13] and [14] are digit serial multipliers. Thus

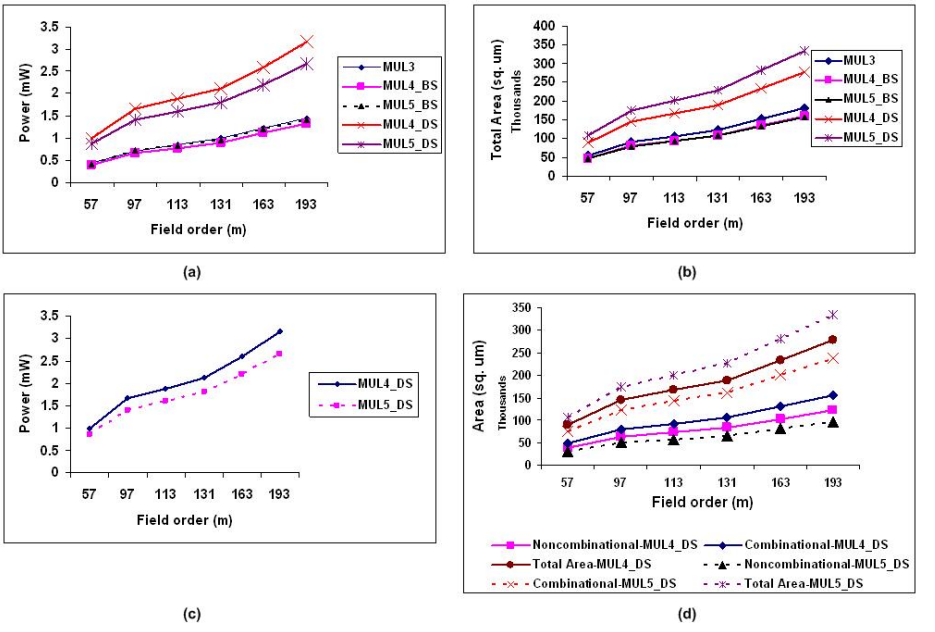


Fig. 6. (a) Power and (b) Area results of bit-serial multipliers (c) Power and (d) Area results of digit-serial multipliers

based on the architectures of [13] and [14], we create four variants, bit-serial multipliers – MUL4_BS, MUL5_BS and digit-serial multipliers – MUL4_DS and MUL5_DS. The delay in all the bit-serial implementations is equal to 1.36 ns as the critical path is $T_{2-AND} + 2T_{2-XOR}$ in all the designs. In contrast to the bit-serial implementations, the delay of digit-serial multipliers MUL4_DS and MUL5_DS is 4.16 ns and 9.18 ns, respectively for a digit length of 8.

The serial multipliers are compared for power and area in Fig. 6(a) and Fig. 6(b), respectively. MUL3, MUL4_BS and MUL5_BS show almost equal power results for all field orders. MUL3 occupies more area than MUL4_BS and MUL5_BS. It is interesting to note that the bit-serial multipliers, MUL4_BS and MUL5_BS, which are the bit-serial derivatives of the digit-serial implementations, outperform MUL3 in terms of area which was designed with an intention to operate as a bit-serial multiplier. As expected, digit-serial implementations are worse off compared to their bit-serial counterparts.

Fig. 6(c) and (d) compare power and area of the two digit serial implementations, MUL4_DS and MUL5_DS. The two multipliers were compared at gate level in [14]. The ASIC implementation of the two designs shows that MUL4_DS is better than MUL5_DS in terms of area but worse off with respect to dynamic power dissipation. In Fig. 6(d), combinational, noncombinational and total areas of MUL4_DS and MUL5_DS are compared. The dotted lines correspond to the area components of MUL5_DS and the solid line curves are used for those of

Table 1. Results for MUL6 and MUL7

Design	Critical path delay (ns)	Silicon area ($\mu\, m^2$)	Dynamic power (mW)
MUL6	29.34	16358291	341.2509
MUL7	29.55	16678856	342.4379

MUL4_DS. Although MUL5_DS has lower non-combinational area, its outweighing combinational area makes it less area efficient than MUL4_DS. This somewhat unexpected synthesis result also explains the higher critical path delay of MUL5_DS (9.18 ns as opposed to 4.16 ns of MUL4_DS).

Generalized bit-level parallel implementations [11]: MUL6 and MUL7 are the generalized bit-parallel implementations of LSB-first and MSB-first algorithms reported in [11] which are implemented for the highest order that is considered in this paper, i.e., M = 193. Table 1 shows the results of the two designs. The results show no significant deviation in all VLSI metrics between the two implementations. If these results are compared with the fixed order bit-parallel architectures of MUL1 and MUL2, the critical path delay is higher. Though MUL6 and MUL7 are configurable in terms of field order, they are an order of magnitude (21 times) slower than their fixed order counterparts due to the vertical critical path discussed previously. The area of MUL6 and MUL7 is also higher than MUL1 and MUL2 (1.77 and 2.72 times higher, respectively) due to the extra circuitry (LCELLs) needed to determine the actual field order m.

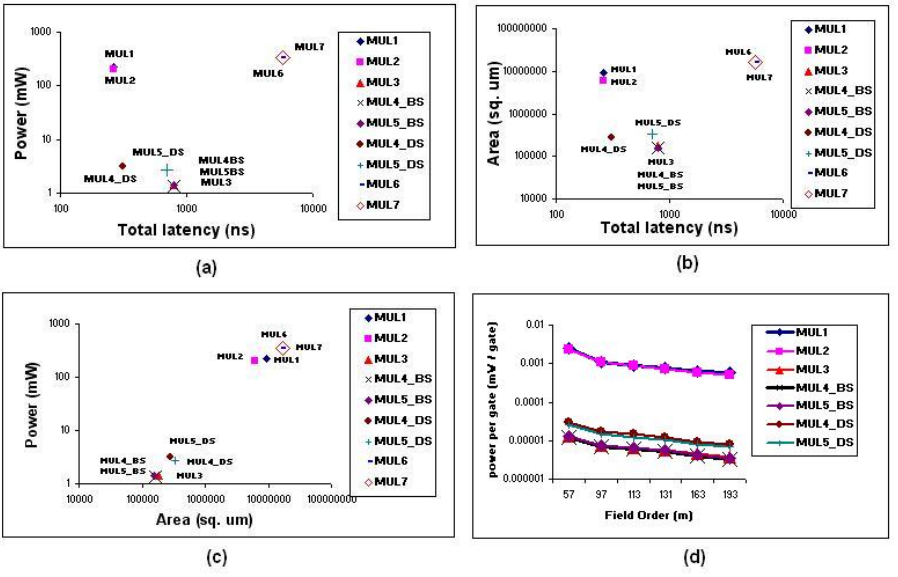


Fig. 7. Consolidated Results

Consolidated results - power vs. total latency: Fig. 7(a)shows a scatter plot of power dissipation against latency of all finite field multipliers discussed in this paper for field order of 193. The axes are scaled logarithmically for clarity. This plot provides the system architect with a choice of finite field multipliers in a large design space to trade off between power and performance. For fair proposition that involves both bit serial and digit serial multipliers, latency instead of worst case delay is used. It refers to the product of the worst case delay and number of cycles to complete a single multiplication. Serial implementations outperform parallel multipliers in terms of power dissipation. Despite having similar critical path delays (1.36 ns) due to the use of similar basic cells, the bit-serial multipliers show thrice as much total latency as the bit-parallel multipliers. This is attributable to the difference in the number of clock cycles ($3m$ clock cycles for bit-serial multipliers and $m + 1$ clock cycles for bit-parallel multipliers) needed to complete one finite field multiplication operation. MUL4_BS has twice the total latency of its digit-serial counterpart, MUL4_DS. However, the total latency of MUL5_BS is around 1.13 times that of MUL5_DS. This contrast between digit-serial multipliers is because of the critical path delays. Generalized bit-parallel multipliers MUL6 and MUL7, have a total latency of 21 times their fixed bit-parallel counterparts MUL1 and MUL3. This is due to the chain of LCELLs used to determine the field order. In terms of power dissipation, parallel implementations consume on average, around 78 times higher power than the serial implementations.

Consolidated results area vs. total latency: Silicon area of each multiplier is plotted against latency in Fig. 7(b). Serial multipliers again occupy the lower region of the graph with digit-serial multipliers occupying higher area than bit-serial implementations. Bit-serial multipliers, MUL3, MUL4_BS and MUL5_BS, on an average use only 1/58 the area of bit-parallel architectures, MUL1 and MUL2. Digit serial multipliers, MUL4_DS and MUL5_DS, occupy about twice the area of their bit-serial designs, MUL4_BS and MUL5_BS. Generalized bit-parallel designs occupy more area than fixed bit-parallel multipliers. The area of MUL6 (LSB-first architecture) is double that of MUL1, and the area of MUL7 (MSB-first architecture) is around 3 times that of MUL2.

Consolidated results power vs. area: Fig. 7(c) shows the relationship between the average power dissipation and area of each implementation. All serial multipliers are cluttered in the low power, lower area region whereas the parallel implementations occupy the other extreme.

Consolidated results power per gate vs. field order: In Fig. 7(d), we show the power consumed per gate against field order for all multipliers except generalized parallel implementations. Power per gate is calculated by dividing the average dynamic power by the number of gates used for the implementation. The total number of gates is determined from the total area divided by the area occupied by one two-input $NAND$ gate in TSMC 0.18μm technology library. Though total power increases as field order increases, power consumed by each gate is lower as the field order increases as evinced by the consistent trends of all designs in Fig. 7(d). The bit parallel multipliers exhibit higher power consumption per gate than the bit

serial or digit serial multipliers. This implies the increased probability of spurious computations in some gates with more parallelism.

5 Conclusion

In this paper, we consolidated some recently reported systolic and semisystolic finite field multipliers. A brief description of algorithm to architecture mapping for each implementation was shown followed by the comprehensive simulations of their ASIC prototypes based on TSMC 0.18μm technology library. We compared their performances based on various criteria such as silicon area, dynamic power dissipation and critical path delay and overall latency. It was shown that bit-parallel architectures consume higher area and power than bit-serial implementations. The two digit-serial architectures in [13] and [14] were also reduced to bit serial variants for analysis. Our evaluation also revealed the agility of generalized bit-parallel architectures to suit varying field order is achieved with a severe penalty of performance and area overhead. With the emphasis on low power design of mobile computing and performance per unit cost of smart card application, this review provides a good insight into the potential design space exploration of the array-type finite field multipliers. Future research would involve studying these architectures to overcome their shortcomings and on their resilience against differential power and differential time attacks when they are used to implement the ECC engine.

References

1. B. Schneier, *Applied Cryptography*. 2nd Edition, Wiley 1996.
2. A. Menezes, *Elliptic Curve Public Key Cryptography*. Kluwer Academic Publishers, 1993.
3. D. Hankerson, J. L. Hernandez, A. Menezes, Software Impelementation of elliptic curve cryptography over binary fields, *Cryptographic Hardware and Embedded Systems*, pp. 1-24, Springer-Verlag, Heidelberg, 2000.
4. H. Eberle, N. Gura, S. Chang-Shantz, A cryptographic processor for arbitrary elliptic curves over $GF(2^m)$, in *Proc. IEEE Intl. Conf. on Application-Specific Systems, Architectures, and Processors*, Hague, Netherlands, pp. 444-454, June, 2003.
5. M.A. Hasan, M.Z.Wang, V.K. Bhargava, A modified Massey-Omura parallel multiplier for a class of finite fields, *IEEE Trans. on Computers*, vol. 42, no. 10, pp. 1278-1280, Oct. 1993.
6. T. Zhang, K.K. Parhi, Systematic design of original and modified mastrovito multipliers for general irreducible polynomials, *IEEE Trans. on Computers*, vol. 50, no. 7, pp. 734-749, July 2001.
7. E.D. Mastrovito, VLSI designs for multiplication over finite fields $GF(2^m)$, in *Proc. Sixth Intl. Conf. Applied Algebra, Algebraic Algorithms, and ErrorCorrecting Codes (AAECC-1988)*, Rome, Italy, pp. 297-309, July 1988.
8. A. Halbutogullari, C. K. Koc, Mastrovito Multiplier for general irreducible polynomials, *Applied Algebra, Algebraic Algorithms and Error-Correcting Codes, Lecture Notes in Computer Science Series No. 1719*, pp. 198-507, Springer-Verlag, Berlin, 1999.

9. A. Reyhani-Masoleh, M. A. Hasan, Low complexity bit parallel architectures for polynomial basis multiplication over $GF(2^m)$, *IEEE Trans. on Computers*, vol. 53, no. 8, pp. 945-958, Aug. 2004.
10. S. K. Jain, K. K. Parhi, Low latency standard basis $GF(2^m)$ multiplier and squarer architectures, in *Proc. IEEE Intl. Conf. on Acoustic, Speech and Signal Processing (ICASSP-1995)*, Detroit, Michigan, USA, pp. 2747-2750, May 1995.
11. S. K. Jain, L. Song, K. K. Parhi, Efficient semisystolic architectures for finite-field arithmetic, *IEEE Trans. on Very Large Scale Intergration (VLSI) Systems*, vol. 6, no. 1, pp. 101-113, Mar. 1998.
12. C.-L. Wang and J.-L. Lin, Systolic array implementation of multipliers for finite fields $GF(2^m)$, *IEEE Trans. on Circuits and Systems-I*, vol. 38, no. 7, pp. 796-800, July 1991.
13. J.-H. Guo and C.-L. Wang, Digit-serial systolic multiplier for finite fields $GF(2^m)$, *IEE Proc. Comput. Digit. Tech.*, vol. 145, no. 2, pp. 143-148, Mar. 1998.
14. K.-W. Kim, K.-J. Lee, K.-Y. Yoo, A new digit-serial multiplier for finite fields $GF(2^m)$, in *Proc. of Info-tech and Info-net (ICII 2001)*, Beijing, China, vol. 5, pp. 128-133, Oct. 2001.
15. C. H. Kim, S. D. Han, C. P. Hong, An efficient digit-serial systolic multiplier for finite fields $GF(2^m)$, in *Proc. of 14th Annual IEEE Intl. ASIC/SOC Conference*, pp. 361-165, Sept. 2001.
16. J. Lopez and R. Dahab, An overview of elliptic curve cryptography, *Technical Report*, Institute of Computing, State Uniersity of Campinas, May 2000.
17. Certicom Research, SEC 2: Recommended elliptic curve domain parameters, *Standards for Efficient Cryptography*, Technical document, Sept. 20, 2000.
18. B. A. Laws, C. K. Rushforth, A cellular-array multiplier for $GF(2^m)$, *IEEE Trans. on Computers*, vol. C-20, pp. 869-874, Nov. 1982.
19. R. Lidl, H. Niederreiter, *Introduction to finite fields and their applications*, Revised Edition, Cambridge University Press, 1994.
20. N. Nedjah, L. de Macedo Mourelle, A reconfigurable recursive and efficient hardware for Karatsuba-Ofman's multiplication algorithm, in *Proc. IEEE Int. Conf. on Control Applications (ICCA 2003)*, Istanbul, Turkey, vol. 2, pp. 1076-1081, June 2003

Analysis of Real-Time Communication System with Queuing Priority

Yunbo Wu[1,2], Zhishu Li[1], Yunhai Wu[3], Zhihua Chen[4], Tun Lu[1], Li Wang[1], and Jianjun Hu[1]

[1] School of Computer Science, Sichuan University, 24 southern section1, 1st Ringroad, ChengDu, China
(ybwutm,tm_001,tm_002,ths_01,juan01_tom)@tom.com
[2] Ningbo Fashion Institute, Ningbo, China
[3] Yunnan Design Institute of Water Conservancy and Hydroelectric Power, Kunming, China
ybwuyh2002@yahoo.com
[4] Kunming Meteorological Administration, west of XiHua Park, Kunming, China
khzitm@tom

Abstract. In this paper, we discuss the real-time issue related to QoS requirements in a communication system, which consists of several sensors and one central control unit. In order to guarantee real-time requirement and fairness among sensors, a polling mechanism with queuing priority is applied in it. By means of the imbedded Markov chain theory and generating function method, mean queue length and mean circle time of asymmetric system are studied quantitatively, and corresponding exact expressions are obtained. These results are critical for system device and system performance analysis. Finally, computer simulations demonstrate effectiveness of our analysis.

1 Introduction

Real-time is a crucial parameter of QoS demands in networking applications. In a bus-based industry control system, several sensors and one central control unit share the communication channel. The sensor collects information of devices and sends them to the central control unit. Also, the sensor receives instructions from the central unit and controls corresponding devices according to them. The central unit must acquire information from sensors and submit controlling instructions to them in time. In this scenario, there are two key issues deserving more attention. One is timely response both in sensors and central unit. Another is fairness among sensors, namely to share the bus channel in fair way. For this purpose, we adopt a polling system with queuing priority as implementation fashion, in which the service discipline to all queues is exhaustive. The central control unit has the privilege of queuing priority, i.e. whenever the central control unit needs to send data, the bus has to serve it as long as the bus is idle. On the other hand, queues of sensors are served in a cyclic order.

Traditional polling system consists of multi-queues and one server. In which, queues are polled by the server in a cyclic order under a specific service policy, such as gated service, exhaustive service, or limited service. Since polling systems being popularly applied in communication network and CIMS(computer integrated make systems), it's

T. Srikanthan et al. (Eds.): ACSAC 2005, LNCS 3740, pp. 707 – 713, 2005.

not surprise that there are many researchers paying more attention on traditional polling systems. However, to obtain exactly results is a much difficult thing under asymmetrical load conditions. Since 80's, lots of researchers have exerted themselves to this problem. Ref. [1] gives analysis to the fundamental symmetric polling systems in detail, and obtains meaningful result of mean queue length and mean queue delay time. While Ref. [2,3] study the asymmetric polling systems and approximate results are obtained. Recently, Ref.[4,5,6] discusses Blue-tooth Piconets scenario via simulation, and some meaningful results of polling system are given.

Unlike classic polling systems above-mentioned, our model is a polling system with queuing priority. For the purpose of performance analysis and system device, mean queue length and mean circle time are discussed in our work. By the method resembling Ref.[1], we use embedded Markov chain and probability generation function method to analyze the system. One of the features of this study is asymmetric system, and the other is discrete-time analysis. The rest of this paper is organized as follows: Section 2 describes mathematical model of the bus-based industry control system. Section 3 discusses the mean queue length and mean circle time of asymmetric system, and corresponding expressions are explicitly obtained. Section 4 gives simulation results, and last section concludes this paper.

2 System Model

The bus-based industry control system is modeled as a polling system depicted in Fig.1, in which the central queue has the privilege of queuing priority. The polling system includes number N normal queues and one central queue C, and has one server to serve all the queues according to exhaustive service policy. The server serves the normal queues according to the cyclic order, i.e. queue 1, queue 2, …, queue N, then queue 1 again. Whenever the central queue has packets to be sent, the privilege of central queue is that if server is right in idle then queue C is served at once, otherwise after finishing current normal queue the server turns to serve queue C, and then the next normal queue in previous sequence is polled after a switchover interval. In other words, this privilege of queue C is non-interrupted priority.

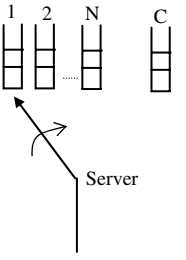


Fig. 1. System Model

In this model, normal queues are related to buffer queues of sensors and queue C related to the buffer of central control unit, respectively. While the server models the bus channel. There are three independent random processes relating to each queue separately, i.e. the packet arrivals process, service process of a packet in a queue, and switching-over process from one queue to another. Considering general application purpose, parameters associated with these random processes are usually not the same. Namely, this queuing model is asymmetric system. In the exhaustive service policy, when the server visits the queue it serves all the packets in the queue, but it doesn't serve packets that arrive during service period of this queue. The exhaustive service can assure the fairness requirement among all sensors, and has been proved to perform better than other service policy in Ref.[6,7].

3 Theoretical Analysis of Model

3.1 Model Analysis

As mentioned above, there are three independent random processes corresponding to each queue separately in our model, i.e. the arrival process, service process, and switchover process. The arrival process at a queue is independent of the arrival processes at other queues. While the service process at a queue is assumed to be independent of the arrival processes at all queues, and of the service processes at other queues. In our analysis, we condition the discrete system and infinite capacity in each queue. Then some characteristics will be presented as following:

a) The arrival process at *queue i* is independent and Poisson distributed with rate λ_i, corresponding probability generating function (PGF) is denoted by $A_i(z)$, and first moment is denoted by λ_i, $i=1,2,\ldots,N,C$.

b) The service time of *queuei* is an independent random variable with corresponding PGF $B_i(z)$, and first moment β_i, $i=1,2,\ldots,N,C$. In addition, packets are served on FCFO basis, and the service policy is of exhaustive.

c) When the server move from *queue i* to *queue (i+1)* a switchover time is incurred, which is general distributed. We let R_i denote the PGF for switching-over time in this case, $i=1,2,\ldots,N,C$.

Throughout this paper, the standard stability is assumed [7]:

$$\sum_{i=1}^{N} \beta_i \lambda_i + \beta_c \lambda_c = \sum_{i=1}^{N} \rho_i + \rho_c < 1$$

We suppose that *queue i, queue (i+1)* ($i=1,2, \ldots ,N$) are serviced at t_n instant and t_{n+1} instant, respectively, while central queue C is serviced at t_n^* instant. Then the system probability generation functions corresponding to instant t_{n+1} and instant t_n^* are deduced from above-mentioned model as following:

Lemma 1. Notation $G_{i+1}(z_1, z_2, \ldots, z_N, z_c)$ represents the joint generating function in steady state at the time t_{n+1} when queue Q_{i+1} is visited by server, then G_{i+1} satisfies following equation:

$$
\begin{aligned}
&G_{i+1}(z_1,\ z_2,\ldots,\ z_N,\ z_c)\\
&=R_{c,i+1}\left(\prod_{j=1}A_j(z_j)A_c(z_c)\right)\cdot R_{i,c}\left(\prod_{j=1}A_j(z_j)A_c\left(B_c\left(\prod_{j=1}A_j(z_j)F_c\left(\prod_{j=1}A_j(z_j)\right)\right)\right)\right)\cdot\\
&G_i\{\ z_1,z_2,\ldots,z_{i-1},B_i(\ \prod_{j=1,\neq i}A_j(z_j)A_c\left(B_c\left(\prod_{j=1}A_j(z_j)F_c\left(\prod_{j=1}A_j(z_j)\right)\right)\right)\cdot\\
&F_i\left(\prod_{j=1,\neq i}A_j(z_j)A_c\left(B_c\left(\prod_{j=1}A_j(z_j)F_c\left(\prod_{j=1}A_j(z_j)\right)\right)\right)\right),z_{i+1},\ldots,z_N,B_c\left(\prod_{j=1}A_j(z_j)F_c\left(\prod_{j=1}A_j(z_j)\right)\right)\}
\end{aligned}
\tag{1}
$$

Where $F_i(x)(i=1,2,\ldots,N,C)$ denotes the PGF of service time when x packets in *queue i* were served under exhaustive service policy.

Proof:

Some random variables used later are presented as follows:

$u_i(n)$: switch-off duration relating to server's moving at t_n instant from queue Qi to queue Q_{i+1},

$v_i(n)$: service time of queue Qi which begins at t_n instant,

$\eta_j(u_i)$: packets arriving in queue Qj during u_i interval,

$\eta_j(v_i)$: packets arriving in queue Qj during v_i interval,

$\xi_i(n)$: packets of queue Qi at t_n instant.

According to the above-mentioned model, packets stay in system queues at time t_{n+1} satisfy following equations:

$$
\begin{cases}
\xi_j(n+1)=\xi_j(n)+\eta_j(u_{c,i+1}+u_{i,c})+\eta_j(v_i+v_c)\\
\xi_c(n+1)=\eta_c(u_{c,i+1})\\
\xi_i(n+1)=\eta_i(u_{c,i+1}+u_{i,c})+\eta_i(v_c)
\end{cases}
\tag{2}
$$

Where $v_c=\tau_c[\ \eta_c(u_{i,c})+\eta_c(v_i)+\xi_c(n)\ ]$, and $\tau_c(x)$ denotes the service time of number x packerts in *queue c* under exhaustive servie policy; $u_{i,c}$ and $u_{c,i+1}$ denote the shifting duration from *queue i* to *queue i+1* and *queue c* to *queue i+1*, respectively, and $u_i=u_{i,c}+u_{c,i+1}$. While $\gamma_{i,c}$, $\gamma_{c,i+1}$, and γ_i are mean value related to random valuables $u_{i,c}$, $u_{c,i+1}$, and u_i, respectively. By means of equetion (2), the PGF can be obtained.

Lemma 2. At time t_n^* when *queue c* is visited by server, the system joint generating function in steady state can be expressed as follows:

$$
\begin{aligned}
&G_c(z_1,\ z_2,\ldots,\ z_N,\ z_c)\\
&=R_{i,c}\left(\prod_{k=1}^{N}A_k(z_k)A_c(z_c)\right)\\
&\cdot G_c\left(z_1,z_2,\ldots,z_{i-1},B_i\left(\prod_{k=1,k\neq i}^{N}A_k(z_k)A_c(z_c)F_i\left(\prod_{k=1,k\neq i}^{N}A_k(z_k)A_c(z_c)\right)\right),z_{i+1},\ldots,z_N,z_c\right)
\end{aligned}
\tag{3}
$$

Proof:

Packets stay in system queues at time t_n^* satisfy following equations:

$$\begin{cases} \xi_c(n^*) = \xi_c(n) + \eta_c(u_{i,c}) + \eta_c(v_i) \\ \xi_i(n^*) = \eta_i(u_{i,c}) \\ \xi_j(n^*) = \xi_j(n) + \eta_j(u_{i,c}) + \eta_j(v_i) \end{cases} \tag{4}$$

Using equation (4), *Lemma 2* can be proved in a way similar to the proof of *Lemma 1*.

3.2 Mean Queue Length

The first moments of system is related to the mean queue length of queues in aforementioned polling system. At the instant that server arrives at *queue (i+1)*, the mean queue length of *queue j* can be calculated as follows:

$$g_{i+1}(j) = \lim_{x_1, x_2 \cdots x_N \to 1} \frac{\partial G_{i+1}(x_1, x_2, \ldots, x_N, x_c)}{\partial x_j}$$

So we can obtain following equations:

$$\begin{cases} g_{i+1}(j) = \gamma_{c,i+1}\lambda_j + \dfrac{\gamma_{i,c}\lambda_j}{1-\rho_c} + g_i(j) + \dfrac{\beta_i\lambda_j g_i(i)}{(1-\rho_i)(1-\rho_c)} + \dfrac{\beta_c\lambda_j g_i(c)}{1-\rho_c} \\ g_{i+1}(i) = \gamma_{c,i+1}\lambda_i + \dfrac{\gamma_{i,c}\lambda_i}{1-\rho_c} + \dfrac{\rho_i\rho_c g_i(i)}{(1-\rho_i)(1-\rho_c)} + \dfrac{\beta_c\lambda_i g_i(c)}{1-\rho_c} \\ g_{i+1}(c) = \gamma_{c,i+1}\lambda_c \end{cases} \tag{5}$$

Where, the switch-off duration server walking from q*ueue i* to *queue(i+1)* is satisfied to $u_i = u_{c,i+1}$, and corresponding mean is $\gamma_i = \gamma_{c,i+1}$. Here, $u_{i,c}$ is so small as to be ignored according to control system; $i, j = 1,2,,...,N$.

Using iterative algorithm to equations (5), mean queue length of normal queues of asymmetric system is obtained as following:

$$g_j(j) = \frac{(1-\rho_i)\lambda_j \sum_{i=1}^{N} \gamma_i}{1-\rho_c - \sum_{i=1}^{N} \rho_i} \left[1 + \frac{\sum_{i=1}^{N} \lambda_i\rho_i - \lambda_j \sum_{i=1}^{N} \rho_i}{(1-\rho_c)\lambda_j} \right] \tag{6}$$

Similarly, the mean queue length of central queue can be derived as well:

$$g_c(c) = g_i(c) + \frac{\beta_i\lambda_c g_i(i)}{(1-\rho_i)} \tag{7}$$

When the system is symmetric, i.e. $\rho_i = \rho$, $\gamma_i = \gamma$, $\lambda_i = \lambda$, the mean queue length of normal queue and central queue are expressed as follows, respectively:

$$g = \frac{(1 - \rho)N\gamma\lambda}{1 - \rho_c - N\rho}$$

$$g_c = \frac{\lambda_c\gamma(1 - \rho_c)}{1 - \rho_c - N\rho}$$

3.3 Mean Circle Time

In our polling model, the circle time refers to the duration that the server visits twice the same normal queue successively. So, mean circle time can be obtained intuitively as follows:

$$\bar{\theta} = \sum_{i=1}^{N}\left(\frac{g_i(i)\beta_i}{1-\rho_i} + \frac{g_i(c)\beta_c}{1-\rho_c} + \gamma_i\right) \tag{8}$$

When system is symmetric, mean circle time is:

$$\bar{\theta} = \frac{N\gamma}{1 - \rho_c - N\rho}$$

4 Simulation Results

Taking as example a symmetrical polling system with N=40 queues having infinite capacities, numerical results are presented in this section to discuss the theoretical analysis and the approximation accuracy. The value of parameters is chosen to be uniform both in theoretical analysis and simulation. The arrival process is Poisson distribution, while service time and switch-off time are General distributions with mean

Table 1. Mean circle time: simulation result vs. theoretical analysis

Total Load $(N\rho+\rho_c)$	Theoretical $\bar{\theta}$ (s)	Experimental $\bar{\theta}$ (s)			
		$\rho_c/\rho=1$	$\rho_c/\rho=2$	$\rho_c/\rho=3$	$\rho_c/\rho=4$
0.1	0.004444	0.004305	0.004412	0.004520	0.004474
0.2	0.005	0.005007	0.005026	0.005102	0.005014
0.3	0.005714	0.005716	0.005722	0.005677	0.005707
0.4	0.006667	0.006672	0.006659	0.006665	0.006683
0.5	0.008	0.008012	0.008101	0.008080	0.007875
0.6	0.01	0.01002	0.01006	0.01015	0.01009
0.7	0.01333	0.01316	0.01323	0.01327	0.01342
0.8	0.02	0.02001	0.02001	0.02007	0.02021
0.9	0.04	0.04007	0.04003	0.04054	0.04008

$\beta = \beta_c = 0.00001s$ and $\gamma = 0.0001s$, respectively. Our simulation experiment is implemented with NS-2 simulator. Simulation results are depicted with their 95% confidence intervals as in Table 1.

The results presented here shows the characteristics that mean circle time $\overline{\theta}$ is affected by the total traffic load $(N\rho + \rho_c)$, where $(N\rho + \rho_c) < 1$. As depicted in Tab.1, the mean circle time increases with total traffic load, while it's seldom influenced by the proportion between normal queue and central queue. Moreover, just as seen in Tab.1, the theoretical analysis results is consistant with simulation's, and this validates our analysis method.

5 Conclusion

In this paper, a bus-based industry multi-sensors control system is being discussed, in which fairness and time-sensitiveness are two important performance criteria. In order to satisfy these requirements, we apply polling mechanism to the control system. Unlike traditional polling system, our polling model privilege the central queue the queuing priority, and the service of each queue is of exhaustive. By Markov chain and probability generation function, we analyze this polling system under asymmetrical traffic load, and obtain exact expressions of mean queue length and mean circle time. The analysis is validated by means of computer simulations. Our analysis results can be used in corresponding control system device and performance assessment.

References

1. Hashida O.: Analysis of Multiqueue the Electrical Communication. Laboratories NTT., 1972,20(3,4):189-199.
2. Mukherjee B, Kwok CK, etc.: Comments on Exact Analysis of Asymmetric Polling Systems with Single Buffers. IEEE Trans on Comm., 1990,COM-38(7):944-946.
3. Porter P G, Zukerman M.: Analysis of a Discrete Mulipriority Queueing System Involving a Central Shared Processor Serving Many Local Queues. IEEE JSAC, February, 1991,(2):88-94.
4. Ka Lok Chan, Vojislav B, etc.: Efficient Polling Schemes for Bluetooth Picocells Revisited. Proceedings of the 37th Hawaii International Conference on System Sciences , 2004.
5. A. Capone, R. Kapoor, etc.: Efficient Polling Schemes for Bluetooth Picocells. In Proc. of IEEE International Conference on Communications ICC2001, volume 7, pages 1990-1994, Helsinki, Finland, June 2001.
6. J. Misic and V.B. Misic.: Modeling Bluetooth Piconet Performance. IEEE Communication Letters, 7(1): 18-20, Jan. 2002.
7. Ibeo O C, Chen X.: Stability Conditions for Multiqueue Systems with Cyclic Service. IEEE Trans Automat Control, 1988, 3(1):102-104.

FPGA Implementation and Analyses of Cluster Maintenance Algorithms in Mobile Ad-Hoc Networks

Sai Ganesh Gopalan, Venkataraman Gayathri, and Sabu Emmanuel

School of Computer Engineering, Block N4, Nanyang Avenue,
Nanyang Technological University, Singapore-639798
gopalansai@pmail.ntu.edu.sg, gayathrivenkat@pmail.ntu.edu.sg,
asemmanuel@ntu.edu.sg

Abstract. A study of hardware complexity and power consumption is vital for algorithms implementing cluster maintenance in mobile ad-hoc networks. This is because of the intrinsic physical limitations of mobile nodes, such as limited energy available, limited computational ability of nodes that form the network. Clustering is divided into two phases, initial cluster formation and cluster maintenance. Cluster maintenance handles situations of change such as a node moving away from a cluster, a new node joining a cluster, clusters splitting due to excessive number of nodes in the cluster, and merging of clusters. In this paper, we have compared the hardware and power efficiency of three cluster maintenance algorithms, Gayathri et al., Lin H.C and Chu Y.H. and Lin C.R and Gerla M. The three algorithms were implemented in synthesizable VHDL to enable porting into FPGA. The hardware complexity and power consumption forms the metrics of comparison of the algorithms studied. For all the algorithms, the CLB slices used was between 123 and 3093 with the operating frequency between 2 MHz and 70 MHz. The total power consumption is between 803 mW and 1002 mW and the total current consumption is between 408 mA and 555 mA.

Keywords: Mobile ad-hoc networks, cluster maintenance algorithm, VHDL (Very High Speed Integrated Circuit Hardware Description Language), FPGA (Field Programmable Gate Arrays).

1 Introduction

A mobile ad-hoc network is an autonomous system of mobile routers (and associated hosts) connected by wireless links. There are no mobility restrictions on these routers and they can organize themselves arbitrarily resulting in unpredictable change in the network's topology. A mobile ad-hoc network may operate in a stand-alone fashion or may be connected to the Internet. The property of these networks that makes it particularly attractive is that they do not require any prior investment in fixed infrastructure. Instead, the participating nodes form their own co-operative infrastructure by agreeing to relay each other's packets [4].

As the size of a mobile ad-hoc network increases, there arises a necessity to develop frameworks that address the scalability issue of the network. Organizing the

T. Srikanthan et al. (Eds.): ACSAC 2005, LNCS 3740, pp. 714 – 727, 2005.

nodes into clusters is one way to handle scalability in ad-hoc networks. Clustering comprises two phases, cluster formation and cluster maintenance.

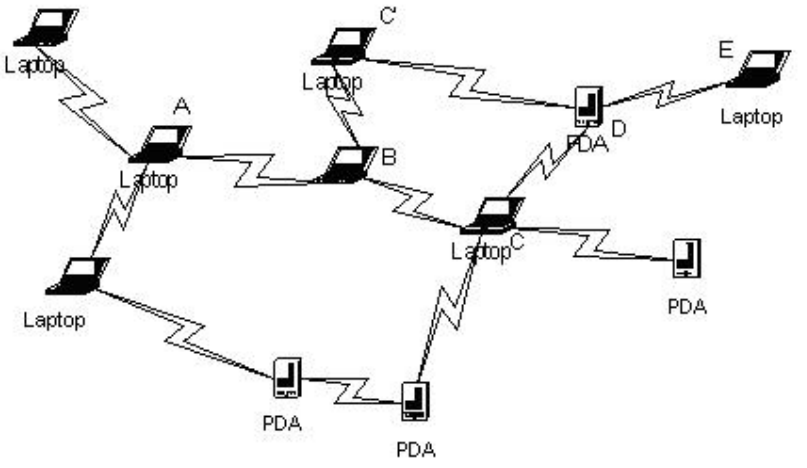


Fig. 1. Ad-hoc network topology

This paper deals with cluster maintenance and assumes that clusters are already formed. In a mobile environment, it may be required that a large ad-hoc network be set up quickly. Partitioning the network into initial groups called clusters forms the basis of cluster formation. In distributed cluster formation, every node participates in cluster formation. This eliminates a single bottleneck or failure point [1]. Cluster formation techniques employ several metrics to form the clusters. Some of the techniques employ number of hops [5], size and number of hops as the metrics [7], while others employ weight as the metric [6].

Once the initial clusters have been formed, the next step is to handle the changing network topology. This is handled by cluster maintenance. Cluster maintenance handles situations of change such as a node moving away from a cluster, a new node joining a cluster, clusters splitting due to excessive number of nodes in the cluster, and cluster merging.

There are several metrics to handle cluster maintenance. Some algorithms use cluster-size restricted maintenance [1], while the other metric popularly used for maintenance is number of hops [2], [3]. Many clustering algorithms have been proposed for MANETs, some clearly dividing their clustering phases, i.e. cluster formation and cluster maintenance [1], [3] while others following a combined approach [2]. It is worthwhile to study the hardware complexity and power consumption of these algorithms for implementation into hardware.

Three clustering algorithms were studied and implemented. These algorithms include “A Novel Distributed Cluster Maintenance Technique for High Mobility Ad-Hoc Networks” [1], “A Clustering Technique for Large Multihop Mobile Wireless Networks” [2], and “Adaptive Clustering for Mobile Wireless Networks” [3]. Out of

the algorithms studied, [1] and [2] are cluster-head based while [3] is non-cluster head based. In each cluster, one node is designated as the cluster-head based on metrics such as lowest ID [2], or based on weight-based election [2]. All three algorithms are distributed in the sense that all nodes are responsible for cluster maintenance.

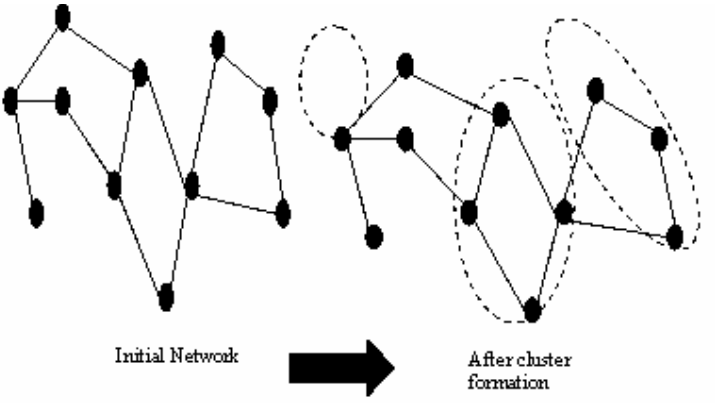


Fig. 2. Initial Cluster Formation

The paper is organized as follows. Section 2 describes the three algorithms studied. Section 3 illustrates the hardware design, while section 4 describes the FPGA implementation. The results are discussed in section 5 and section 6 forms the conclusion of the paper.

2 Background

Algorithm [1], proposed by Gayathri et al. is a distributed algorithm whose main feature is to limit the number of nodes within a cluster. Each node has a state. The state of a node can be identified by six fields, cluster ID C_{id}, node ID N_{id}, cluster head ID CH_{id}, node count in a cluster N_c, time stamp t , cluster size limit in terms of number of nodes in the cluster N. State of a node: $\{C_{id}, N_{id}, CH_{id}, N_c, t, N\}$.Each node can be uniquely identified in the network as a combination of its cluster ID appended with the node ID, which is dynamically allocated when a node joins a cluster. The dynamic allocation of ID is more preferred than globally unique fixed ID, as dynamically allocated ID length is dependent on the number of active devices in the network. The cluster head ID is the node ID of the cluster head in the cluster. Every node in the cluster has information about its cluster head so that it can communicate across the cluster. Every cluster head maintains a cluster head information table wherein the information about other cluster heads is stored. Each node maintains node count N_c so that it does not exceed the cluster size limit in terms of number of nodes N. The cluster size limit in terms of number of nodes in a cluster is used to limit the number of nodes inside a cluster. The time stamp is used to find the time of entry of each node in a cluster. In addition to these features the algorithm tackles cluster

maintenance as a set of scenarios, each of which is handled individually. These can be summarized as a set of messages sent and received. These messages include messages passed when a node joins a cluster, messages passed when a node leaves a cluster, and messages passed when a cluster splits or merges. The drawbacks of this algorithm are that it is quite hardware complex due to the high number of messages passed and the logic involved in the handling of each of these messages. On the other hand, the algorithm tackles all possible scenarios, thereby providing a comprehensive approach to cluster maintenance.

Algorithm [2]**,** proposed by Lin H.C and Chu Y.H., is a hop based clustering algorithm that has combined both cluster formation and cluster maintenance. Within each cluster, nodes are organized within a given number of hops, *R*, called the cluster radius, from the cluster head. When a node moves away from its cluster head and the distance between the cluster head and itself is larger than *R*, it finds another cluster to join otherwise it forms a cluster of its own. Within the network, the ID of each node is unique. Each node maintains a table of information about itself and its neighboring nodes. This information of a node is called the cluster information has four fields, namely node ID *I*, cluster ID C_{id}, distance between itself and its cluster head in number of hops *D* and the ID of the next node in the path to its cluster head N_{id}. Cluster information: $\{I, C_{id}, D, N_{id}\}$. Clusters are merged when the distance between two cluster heads is less than or equal to a predefined number of hops, *D*, called the cluster dismiss distance. The cluster with the larger cluster-head ID is dismissed and the nodes in the dismissed cluster find new clusters to join. This algorithm performed the worse in terms of hardware and power efficiency as compared to the other algorithms because of several reasons. The first was that every time new cluster information was received from a node next on the path of a particular node, all computation of distances to known cluster heads had to be done again. Secondly, this algorithm also calls for the maintenance of tables of information, which in hardware is extremely bulky.

Algorithm [3], proposed by Lin C.R and Gerla M., is a distributed one based on number of hops [3]. This algorithm is a non cluster-head based algorithm and hence is fully distributed. Within each cluster, nodes can communicate with each other in at most two hops. This does not restrict the cluster size in terms of the number of nodes as compared to [1] and hence performance may deteriorate with increase in number of nodes per cluster. Within the network, the ID of each node is unique. Each node also maintains information about its one-hop neighbors. Clustering is split into distinct cluster formation and cluster maintenance phases like [1] and unlike [2]. Cluster maintenance, in this algorithm, is divided into two steps:

Step 1: Check if there is any member of a particular node's cluster that has moved out of its locality.
Step 2: If yes, then based on whether the highest connectivity node is a one hop neighbor or not, remove that node that has moved away or change clusters respectively.

The algorithm does not handle merge and split scenario and performs better in terms of hardware complexity and power consumption as compared to [1] and [2].

3 Hardware Design

The workflow starts with top-down design technique that divides the main module into smaller manageable modules. Then for each of the module listed in the top-down design diagram is implemented in VHDL. Here, we have followed FSM design techniques, which enable better optimization of the circuit during synthesis. We have also reduced the usage of if-then-else VHDL control constructs to a minimum in an effort to reduce the hardware complexities. Verification is carried out by simulation using the test bench. Synthesis is carried out after all codes are verified. After synthesis, placement and routing for the FPGA device is carried. Optimization is carried out after placement and routing to make the module more efficient in terms of hardware complexity and power consumption.

In hardware, only one node is implemented for each algorithm. This is different from the software simulation where the entire nodes in the network are modeled and simulated. This is because in the hardware, we are only interested in the hardware complexity and power consumption analysis of a node. Top-down design in VHDL is a divide-and-conquer strategy where a complex design is divided into smaller reusable design.

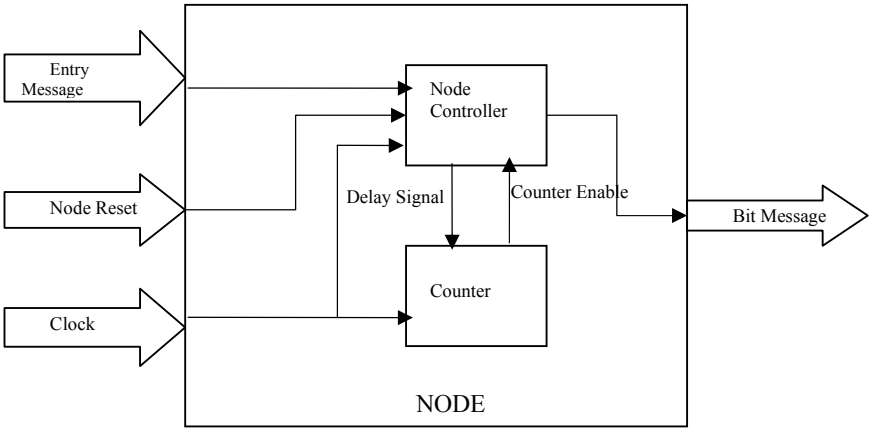


Fig. 3. Generic Block Diagram of Algorithms

The system comprises a top level module called the node which is broken down into a controller that handles all the actual processing of the messages using the finite state transition diagrams shown below and a counter, used for providing a delay circuit. Each message received by a node has to be processed. A certain amount of time is needed to process the received message. In certain cases, analyses of certain cumulative set of messages such as reply messages have to be done. Hence, a module to tell when to analyze in such situations is needed. This sub-module is the counter, which provides the timeout signal in situations when the particular messages are received, after a particular duration of time, as required by the algorithms. Lin C.R

and Gerla M. [3], alone does not make use of the counter sub-module. The state transition diagram of Lin et al. [2] is shown in figure 4.

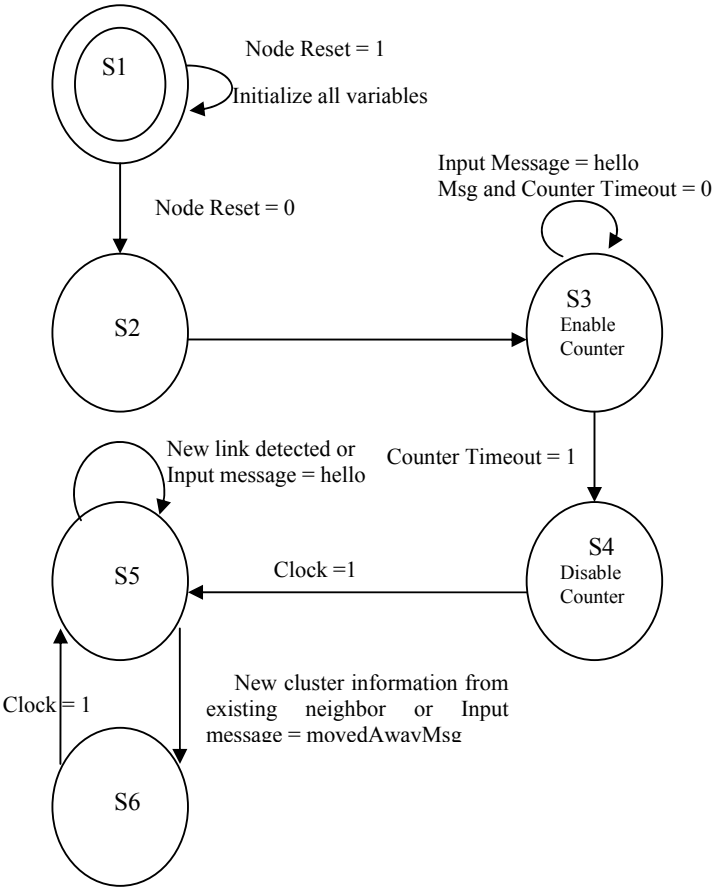


Fig. 4. FSM Diagram for Lin et al. [2]

The messages that are passed in this algorithm [2], are four types: HELLOMSG, REPLYMSG, CLUSTERINFOMSG and MOVEDAWAY message. The cluster radius and the cluster dismiss distance are modeled as global constants. In state S1, the data structures like the information table, own node data, used in the algorithm is initialized. These represent the state of the node after cluster formation. The system remains in state S1 till the node-reset signal remains active. In state S2, the node sends out HELLOMSG to discover its neighbors. In state S3, the node waits for REPLYMSG from its neighbors, as well as sends out REPLYMSG to the HELLOMSG it receives from its neighbors as part of neighbor discovery. Once this is done, the information table is analyzed in state S4 and if nodes which are not in the

initialization phase exist in the information table, then the cluster with the least distance less than cluster radius, between self and cluster head is chosen as the cluster to join. Otherwise, if no neighboring node exists or all neighboring nodes are in initialization phase or no cluster satisfies the cluster radius condition, a new cluster is set up with cluster head as self. In this state, CLUSTERINFOMSG is sent out with the relevant cluster. In states S5 and S6 new CLUSTERINFOMSG are handled.

The state transition diagram for Gayathri et al. [1] is shown in figure 5. The algorithm starts with state S1 where the data structures used in the algorithm are initialized. The node remains in state S1 till the node-reset signal remains active. In state S2, the node accepts various categories of messages and sends control to various states as given in the state transition diagram. The messages accepted include JOINREQ (request for joining a cluster), JOINACC (acceptance message sent by node requesting a join to a cluster), JOINREP (reply sent by node of cluster to which join is requested, to node requesting join), and JOINCONFIRM (confirm message sent to node requesting join to a cluster. Message contains the new node ID assigned to the node), LEAVE (leave message indicating desire to leave a cluster), LEAVEACK (acknowledge message sent to node as a reply to a LEAVE message), HELLOMSG (message sent to discover neighbors as part of a dynamic neighbor discovery process to calculate degree difference when a new cluster head has to be elected), ELECTNEWCLUSTERHEAD (election of new cluster head message containing the node weight calculated according to fixed point arithmetic), SPLITREQ (request by a group of nodes to leave a cluster), MERGEREQ (request to merge with a cluster), MERGEACC (acceptance of a merge request), MERGEEXCHANGE (information exchange messages sent when several merge accept messages are received to decide which cluster to merge with), TOHEADFORMERGE (message sent to cluster head with information about which cluster to merge with at the end of merge exchange process).

State S3, S6 and S15 handle the scenario of a NEWARRIVALBROADCAST (message sent inside cluster indicating new arrival). States S4 and S5 handle all the reply messages obtained by a node that broadcasts a join request, to send out a join accept message to the selected cluster. State S7 is used by the node to synchronize its information with the rest of the cluster. States S8, S9, S10, S11 and S12 handle the election of new cluster head in the event that the cluster head leaves a cluster). State S13 and S14 handle the merging of two clusters.

Gerla et.al [3] also starts with state S1, where locality information, cluster ID, node ID and degree are initialized, representing the state of the system after a cluster formation phase. The node remains in state S1 till the node-reset signal remains active. In state S2, the node sends out CHECKLOCALITYMSG (message sent to check whether all members of the locality are still in the locality, i.e. within 2 hops). State S3 handles receipt of such CHECKLOCALITYMSG and if a particular node is still in the locality of the locality check initiating node, it sends out a STILLINLOCALITYMSG and passes this CHECKLOCALITYMSG with the TTL (time to live) parameter incremented by 1 (in state S4). In every node, when a CHECKLOCALITY message is received, if the TTL parameter is less than 2 (number of hops) then the node is still in the locality of the initiator. Otherwise, an

EXPIREDMEMBERMSG (message indicating member node out of locality). States S3 and S5 also handle EXPIREDMEMBERMSG, according to the algorithm. The state transition diagram for Lin C.R and Gerla M. [3] is shown in figure 6.

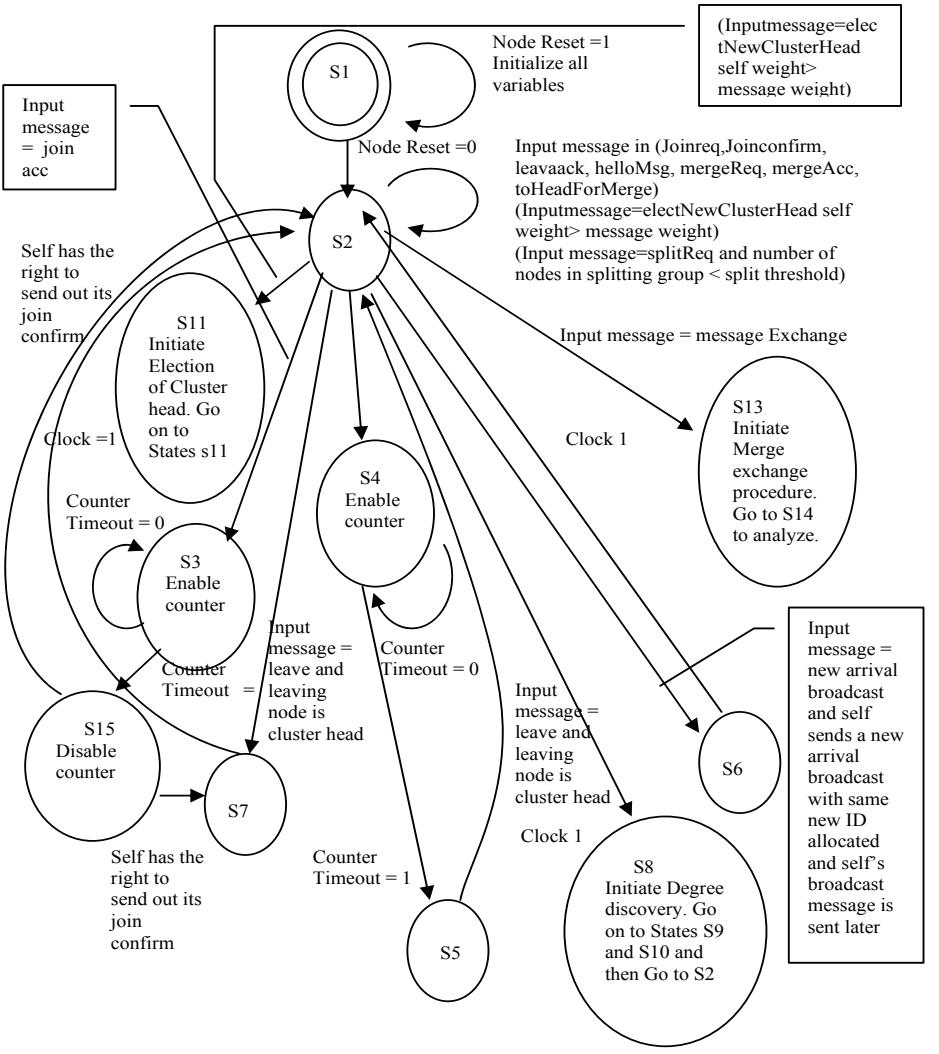


Fig. 5. FSM Diagram for Gayathri et al. [1]

4 FPGA Implementation

FPGA implementation was carried out by building the smallest module first and then proceeding up. Module in the lowest level of the top-down design diagram is implemented and tested first. Module in the top level of the top-down design diagram

is the last module to be implemented and tested. Based on the bottom-up approach, counter and controller modules were implemented and tested first before the node module. Once all the VHDL codes were written, test bench was created for each module, counter, node controller and node. Simulation was carried out in *ModelSim* and the results verified for the functional correctness of the top-level design in relation to other nodes in the network. Input and output signals of each module were observed and verified. Once verified for the correctness, all the VHDL codes were synthesized with LeonardoSpectrum. The FPGA device target is ***Xilinx Virtex-II Pro XC2VP20 -7 FG 676***. Power consumption was estimated with XPower. The design of the circuit created in the synthesis process is used by this estimation to provide a reasonable result of the power consumption.

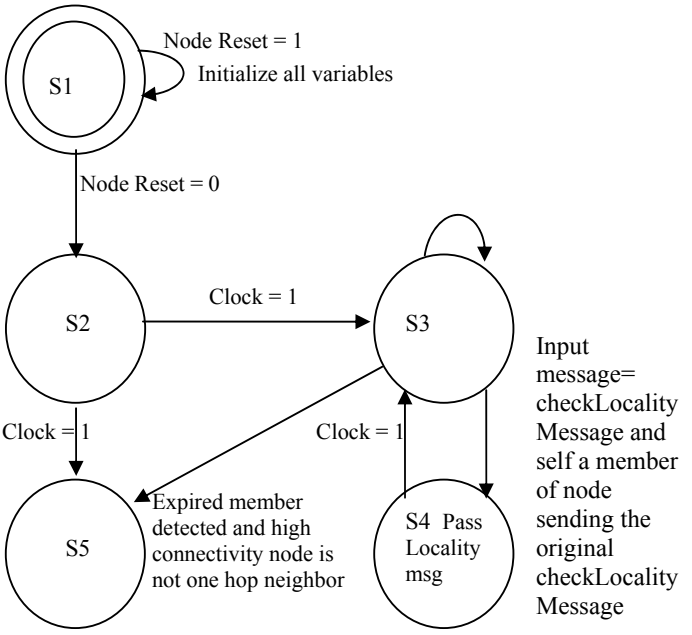


Fig. 6. FSM Diagram for Gerla et al. [3]

5 Results and Discussions

Hardware Complexity

Hardware complexity results can be acquired through placement and routing of the design by using *LeonardoSpectrum*. The FPGA device target is ***Xilinx Virtex-II Pro XC2VP20 -7 FG 676***. An important design consideration is the maximum operating frequency of the device under test. Maximum operating frequency can be obtained from critical path analysis, which analyses the total time taken for an input

signal to propagate to the output. The route taking the maximum time decides the critical path and hence determines the maximum operating frequency of the device under test. The hardware complexity of the three algorithms can be seen from the graph below.

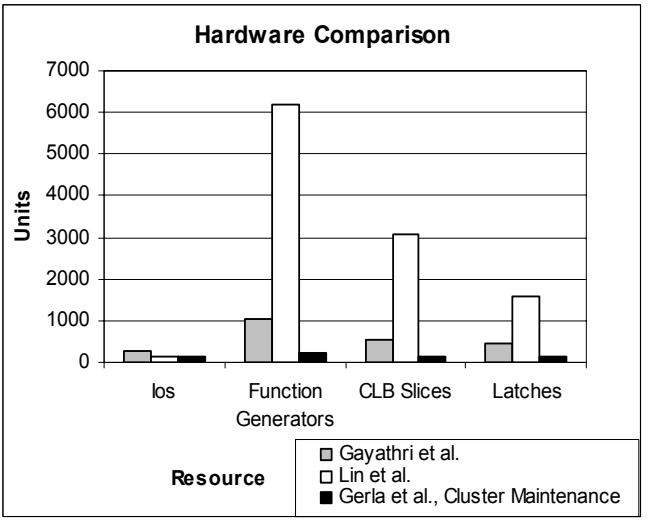


Fig. 7. Hardware comparison of Algorithms

In terms of hardware complexity, Lin H.C and Chu Y.H. [2] had to maintain a table of information of all its neighboring nodes. The storage and indexing of this table in hardware made this algorithm the most complex one in terms of hardware. Further, Lin's algorithm involved complex conditional statements. The complexity of the algorithm arose from the fact that every time a new cluster information was received from a node j which was next in path to the cluster head for the receiving node i, node i had to recalculate the distance between itself and all the cluster heads known to it from the table of information that it had to maintain. Lin C.R and Gerla M. [3] maintenance performed the better than Gayathri et al. [1] and Lin H.C and Chu Y.H. [2]. This was because the maintenance algorithm was extremely simple. The only computation required in this algorithm was the checking of a node moving out of another nodes locality. Gayathri et al. [1] performed better than Lin H.C and Chu Y.H. [2], because the amount of information stored and the processing involved for maintenance were not as complex as Lin H.C and Chu Y.H. [2]. It performed worse than Lin C.R and Gerla M. [3] maintenance in terms of hardware, because this algorithm handled more scenario than [3] maintenance part. Further the number of messages passed in Gayathri et al. [1] was much greater than in Lin C.R and Gerla M. [3] maintenance part as can be seen from the previous chapter. A clear partition of scenarios in Gayathri's algorithm, justifies the use of a greater number of messages as compared to the other algorithms. Finally, the maximum operating frequencies

obtained from the critical path analysis are shown in the diagram below. Once again we see that Lin C.R and Gerla M. [3] maintenance, could operate at higher frequencies than the rest of the two algorithms.

Power Consumption:

FPGA is powered through:

- Vccint: supplies voltage to the core of the device under test (DUT).
- Vccaux: supplies voltage to the Vaux header and Vaux DUT pins and is used to power the JTAG (Joint Test Action Group) boundary pins.
- Vcco: supplies I/O voltages to the DUT.

Power consumption was calculated using XPower tool. The power estimated consists of two parts: quiescent power and dynamic power. Quiescent power is the power that the device consumes when it is idle. This kind of power is extremely important in FPGAs as it is required to feed the transistors that make the programmable logic blocks configurable, whether the device is idle or not. Quiescent power makes up a significant portion of the power consumed by an FPGA chip. Dynamic power is the power consumed by the design due to switching activity. Each element that can switch (LUT, Flip-flops, routing segments etc.) has a capacitance model associated with it.

The power consumed by each switching element in the design is calculated as given below.

$$P = C * V^2 * E * F$$

Where P = Power in Watts, C = Capacitance in Farads, V = Voltage, E = Switching activit , (average number of transitions/clock cycle), F = Frequency in Hz.

Capacitances are determined and fixed during the characterization of the routing resources and the elements needed for a specific device. The voltage is a device specific value fixed by XPower. F * E is the activity rate of each signal. Default activity rates can be set using XPower. Capacitance value used in the power simulation was 35 pF. This is device specific. Voltage is also a device specific fixed value where Vccint is 1.5V, Vccaux and Vcco are 2.5V. Switching activities used in simulation were 10%, 50%, 70% and 100%. Results for 10% switching activities are only analyzes since the other three activity percentages showed the same trend in results. The current and power consumption of the three algorithms can be seen from the graphs below.

From the graphs, we can clearly see that Lin H.C and Chu Y.H. [2] consumed the most amount of current and power, while Gayathri et al. [1] and Lin C.R and Gerla M. [3] were close to each other in their in their power consumption. Current and power consumption of Lin H.C and Chu Y.H. [2] was higher than the rest because it was more complex in terms of hardware. This relates to greater switching activity and hence higher power consumption. The quiescent current and power consumptions are not shown in the graph because the value is constant and same for the FPGA device.

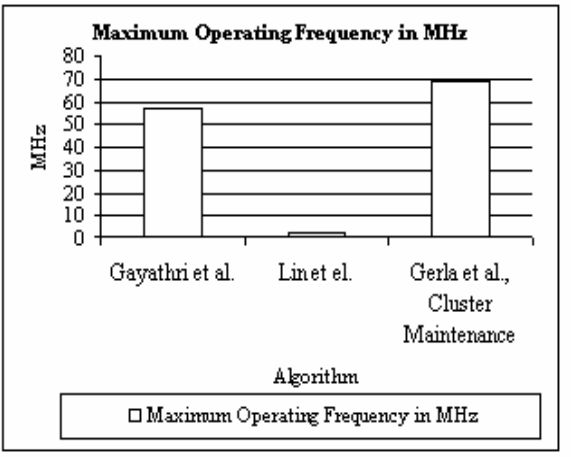


Fig. 8. Maximum Operating Frequency Comparison

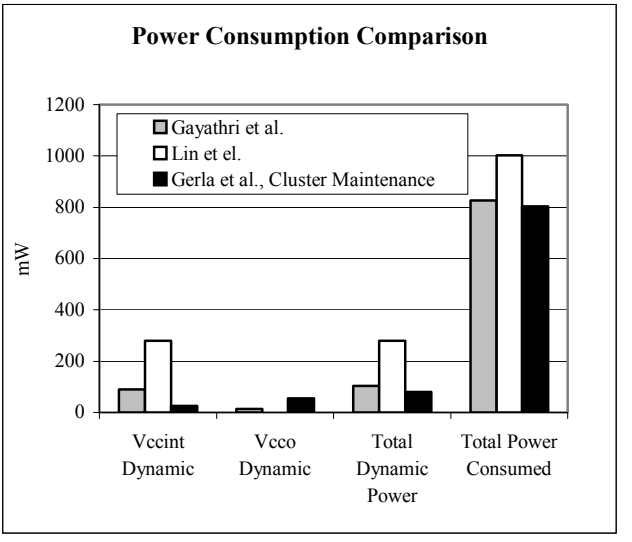


Fig. 9. Power Consumption Comparison

From the discussions above, we see that Lin H.C and Chu Y.H. [2], performs worse both in terms of hardware complexity as well as current and power consumption as compared to the other two algorithms. Gayathri et al. [1], performs slightly worse as compared to Lin C.R and Gerla M. [3] in terms of hardware which is justified considering that it handles more scenarios as compared to Lin C.R and

Gerla M. [3]. Both Gayathri et al. [1] and Lin C.R and Gerla M. [3] perform almost equally in terms of current and power consumption.

Table 1. Quiescent current and power of FPGA device

QUIESCENT	CURRENT (MA)	POWER (MW)
Vccint	200	300
Vccaux	167	418
Vcco	2	5

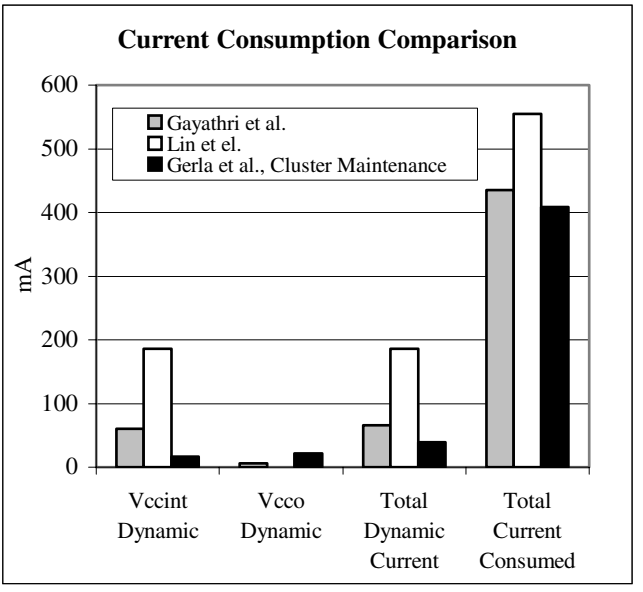


Fig. 10. Current Consumption Comparison

6 Conclusions

The main purpose of this paper is to analyze the hardware complexity, current and power efficiency of the cluster maintenance algorithms. With respect to hardware complexity, Lin C.R and Gerla M. maintenance performed better than the other algorithms with the number of CLB slices it used being 123. Gayathri et al used 529 CLB slices while Lin H.C and Chu Y.H. used 3093 CLB slices. With respect to total power and current consumption, Lin C.R and Gerla M. maintenance and Gayathri et al. were close in their consumptions, with the values being 803 mW, 408 mA for Lin C.R and Gerla M. and 827 mW, 435 mA for Gayathri et al. Lin H.C and Chu Y.H. consumed 1002 mW and 555mA. The results also show that Gayathri et al. and Lin C.R and Gerla M. may be implemented in hardware to suit the nature of devices in

typical wireless networks, while algorithm This is because algorithms Gayathri et al. and Lin C.R and Gerla M. satisfy the simple hardware requirement and low power consumption.

References

1. Gayathri Venkataraman, Sabu Emmanuel and Srikanthan Thambipillai, "A Novel Distributed Cluster Maintenance Technique for High Mobility Ad-Hoc Networks", Proceedings of First International Symposium of Wireless Communication Systems (ISWCS 2004), Mauritius, September 2004.
2. Lin H.C and Chu Y.H., "A Clustering Technique for Large MultiHop MobileWireless Networks", Proceedings of the IEEE Vehicular Technology Conference, May 15 – 18, 2000.
3. Lin C.R. and Gerla M., "Adaptive Clustering for Mobile Wireless Networks", IEEE Journal on Selected Areas in Communications, Vol. 15, No 7, pp. 1265-1275, September 1997.
4. Pahlavan Kaveh and Krishnamurthy Prashant, Principles of Wireless Networks, Pearson Education, 2002.
5. Chen.G, Nocetti.F, Gonzalez.J, Stojmenovic.I, "Connectivity-based K-hop clustering in wireless networks", Proceedings of the 35th Annual Hawaii International Conference on System Sciences -HICSS'02.
6. Chatterjee M, Das S.K, Tugut.D, "WCA: A Weighted Clustering Algorithm for Mobile Ad-hoc Networks", Kluwer Academic publications. 2002.
7. Gayathri V., Emmanuel S., "Size and Hop Restricted Cluster Formation in Mobile Ad-hoc Networks". Proceedings of the IASTED conference of Network and Communication Systems (NCS 2005), Thailand, April 2005.

A Study on the Performance Evaluation of Forward Link in CDMA Mobile Communication Systems

Sun-Kuk Noh

Dept. of Radio Mobile Communication Engineering, Honam University,
59-1 Seobong-dong, Gwangsan-gu, Gwangju, Republic of Korea
nsk7070@honam.ac.kr

Abstract. In the CDMA cellular mobile communication systems, the connection between base and mobile station has a forward link from base station to mobile station. Four channels of forward link, pilot channel, sync channel, paging channel, traffic channel, have an absolute influence on forward coverage and speech quality. And overhead channels(pilot, sync, paging) are assigned output at the adequate rate to optimize the structure of coverage and to minimize the influence of neighbor base station. And the quality of forward link depends on the size of received field strength, as well as Ec/Io which includes both self signal and all noise factors affecting self signal is the main standard in order to decide the coverage and quality of forward link. In this paper, in order to improve a performance of forward link systems in the mobile communications using CDMA, I examine influencing factors in forward links, and measure at LAB., and analyze their results. As the results, I confirm that identify their effect.

1 Introduction

Since mobile communication service using CDMA has started, technical development and changes have followed. CDMA cellular mobile communication service needs to offer good QoS(quality of service) and optimum cell coverage. Various technologies have been developed to improve the QoS and service of mobile communications.[1]-[5] The technical and new service development which aims at more economical and convenience service will maximize the use of frequency spectrum, improve the performance of system, and reduce the cost of building a system and maintenance. DS/CDMA(Direct sequence code-division multiple-access) systems have been used for digital mobile cellular systems and PCS(personal communication services).[6,7] On the other hand, en economical approach to enhance spectrum efficiency is to develop a better cellular engineering methodology. This approach is economical in the sense that it minimizes the cost of BS(base station) equipment. Cellular engineering includes three major aspects : 1) enhancing frequency planning to reduce interference, 2) selecting a cell architecture to improve the coverage and interference performance, 3) choosing better cell site locations to enhance service coverage.[8]

One way of accomplishing this, cell sectorization and improvement of the Ec/Io (Chip Energy/Others Interference). The cell sectorization techniques are

T. Srikanthan et al. (Eds.): ACSAC 2005, LNCS 3740, pp. 728 – 735, 2005.

widely in cellular systems to reduce co-channel interference by means of directional antennas and improvement of the Ec/Io of the pilot channel in forward link, can enhance the speech quality and service of mobile communications and increase the channel capacity, by raising the frequency use efficiency. Furthermore, the costs associated with network construction and operation can be reduced.[9]

In this paper, in order to improve a performance of forward link systems in the mobile communications using CDMA, I examine influencing factors in forward links, and measure at LAB., and analyze their results. As the results, I confirm that identify their effect.

2 Forward Link Systems

In the CDMA mobile communication systems, the connection between BS and MS (mobile station) is divided into forward link from base station to MS and reverse link from MS to BS.[10] In Fig 1, Forward channel includes pilot, sync, paging, and forward traffic channel. Reverse channel consists of access channel and reverse traffic channel.[11,12]

Four channels of forward link have an absolute influence on forward coverage and speech quality. Of these, traffic channel decides its number of channel in proportion to telephone traffic of its BS and BS's maximum output is decided considering the number of traffic channel and the coverage of BS.

Most of the whole output in a BS is assigned to forward traffic channel and overhead channels(pilot, sync, paging) are assigned output at the adequate rate to optimize the structure of coverage and to minimize the influence of neighbor BS. Besides, the output of pilot channel becomes the standard for deciding the coverage of BS and the quality of forward link.[13,14] And the quality of forward link depends on the size of received field strength, as well as Ec/Io which includes both self signal and all noise factors affecting self signal is the main standard in order to decide the coverage and quality of forward link.

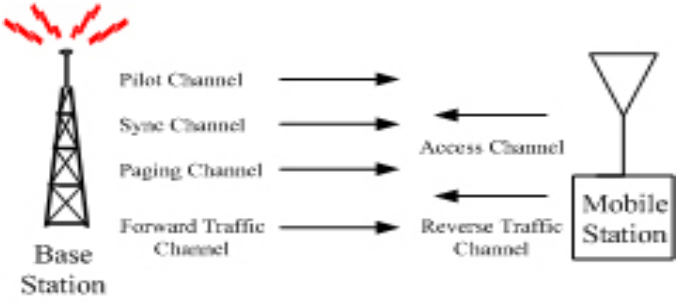


Fig. 1. CDMA link channels

2.1 The Output of Forward Channel

Forward channel is divided by type and digital signal processing in a relevant CHC(channel card) is first modulated by sector by means of intermediate frequency in SICA(Sector Interface Care Assembly). It is modulated in the next XCVU(Up Converter) and inputted to an antenna via LPA. Then the output size of each forward channel is adjusted to a certain level by means of D/G(digital gain) of relevant channel in CHC shown in Fig. 2. D/G has voltage unit on the level of 0~127 in ASIC chip of channel card and Qualcomm recommend the D/G of overhead channel that 108(pilot), 34(sync), 65(paging). Because power is equivalent to the square of voltage, if voltage is increased twice by D/G, power will be increased four times, thereby the channel power on Walsh area is increased as much as 6dB. The output of SICA is converted into wireless transmit frequency band in XCVU and after the output of the whole channel once adjusted to Tx_gain value, it is finally expanded in LPA and copied through an antenna.[15-17]

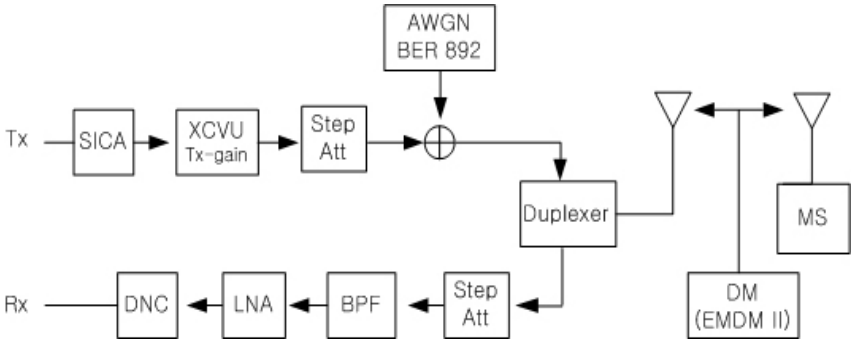


Fig. 2. The schematic diagram of forward link system of CDMA base station

2.2 Receiving Power of MS

MS that moves within the service area of BS measures the Ec/Io and Rx of serving BS regularly and decides hand-off into target base station. Ec/Io and Rx is

$$\frac{Ec}{Io} = \frac{Ec}{Isc + Ioc + NoW} = \frac{Ec}{Rx}[dB] \tag{1}$$

Ec : Energy of pilot signal per chip,
Rx : Total receive power of mobile station
Isc : Interference of self cell(base station),
Ioc : Interference of other cell(base station),
NoW : thermal noise power density

If it is assumed that there is no thermal noise and interference of other BS in equation (1), Ec/Io and Rx are

$$\frac{Ec}{Io}[dB] = 10\log\frac{Pilot\ \ power}{Power\ of\ all\ receive\ signal\ and\ noise} = 10\log\frac{Ppi}{Rx} \quad (2)$$

$$Rx[dBm] = 10\log[P_{PC} + P_{SC} + P_{PgC} + N \cdot P_{TC}] \times 10^3 - Wireless\ \ Pathloss \quad (3)$$

P_{PC} : Pilot channel Power, P_{SC} : Sync channel Power
P_{PgC} : Paging channel Power, N : Traffic channel Number
P_{TC} : Traffic channel Power

3 LAB. Measurements and Results Analysis

3.1 Facotr for LAB. Measurements

Since Ec/Io measures the strength of and the density of interference to pilot channel, after MS in the service area of BS measures Ec/Io, if the value meets the condition compared to T_ADD(Threshold Add) fixed by the system, the mobile station will report to neighbor BS that hand-off is needed, ask hand-off, and changes service BS to maintain call. Then if the condition does not meet compared to T_DROP, it will cut off the lines of all BS. This Ec/Io is the main standard in order to decide the coverage and quality of forward link. S/N(Signal to Noise) becomes the standard to find original signal which is mixed with noise in order to analyze in wireless network. This S/N ratio is expressed as Eb/No(call channel bit energy/noise density) and as Ec/Io in digital system.

If wireless path loss is almost uniformed, the factors influencing on forward link from base station to mobile station may be divided as follows by equation (2) and (3).

1) the power of pilot signal
2) the increase in user (traffic channel)
3) the influence of neighbor base station & of several signals(noise, etc)

For the above 3 major factors, test environment was established in order to measure how each factor affects individually and the influence of the changes of factors on wireless link was examined using mobile station and diagonal monitor(DM).

3.2 LAB. Measurements Environment

For establishing measurements environment, as shown in Fig. 3, the cable connected to LPA in a sending route is linked to step attenuator to remove the influence of LPA leakage power and forward and reverse is balanced with step attenuator.

QCP-800 terminal of Qualcomm which is in a wait state and EMDM-II of Willtec were used to measure the signal between BS and MS (terminal). To generate artificial external noise, BER-892(Bit Error Rate-892) was used as AWGN(Additive White Gaussian Noise) equipment.

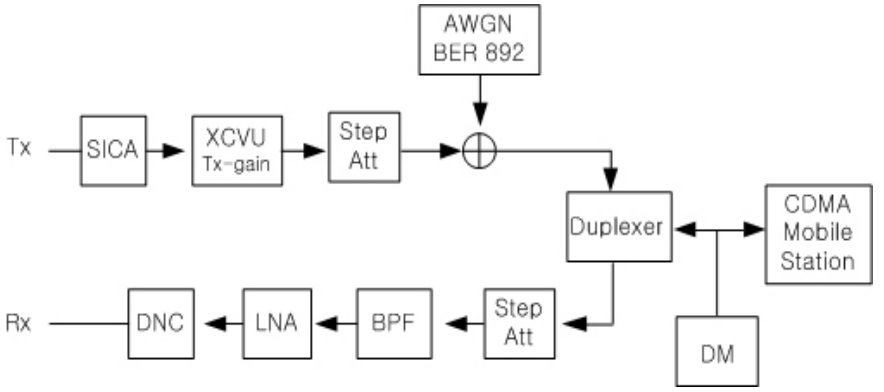


Fig. 3. The schematic diagram of measurements environment of forward link

3.3 LAB. Measurements and Results Analysis

(1) Ec/Io according to pilot channel

1) Measurements and results

To examine Ec/Io of forward link by the changes in forward link channel, the changes in Ec/Io was measured by changing the output of pilot channel which is the quality standard of forward link. Then the D/G of sync channel and paging channel was measured in the fixed state as 34 and 65 respectively. The results are shown Table 1.

Table 1. The comparison of Ec/Io according to the changes in D/G of pilot channel

Pilot D/G	Rx(dBm)	Ec/Io(dB)
93	-71.28	-14.18
98	-71.22	-13.65
103	-71.24	-12.47
108	-71.22	-12.08
113	-71.18	-11.71
118	-71.15	-11.32
123	-71.14	-10.92

2) Results analysis

In Table 1, the changes of D/G in pilot channel have a great influence that Ec/Io is increment.

(2) Ec/Io and Rx according to user(traffic channel)

1) Measurements and results

When it is said about the capacity of CDMA system, it is usually explained as terminal user's interference quantity in a band. Thus the number of terminal used in

relevant base station affects Ec/Io of forward link. The results of measuring Rx and Ec/Io by terminal number using a base station to set D/G of 108(pilot), 34(sync), 65(paging) are Fig. 4.

2) Results analysis

As shown in Fig 4, as the number of user increases, while Rx increases 4.2dBm by 29 subscribers from -36.5dBm to -32.3dBm, Ec/Io deteriorates by 7.69dB from -2.55dB to -10.24dB. The fact that the deterioration of Ec/Io is more serious than the increase in Rx suggests that the increase in telephone traffic principally causes the deterioration of the quality of wireless link.

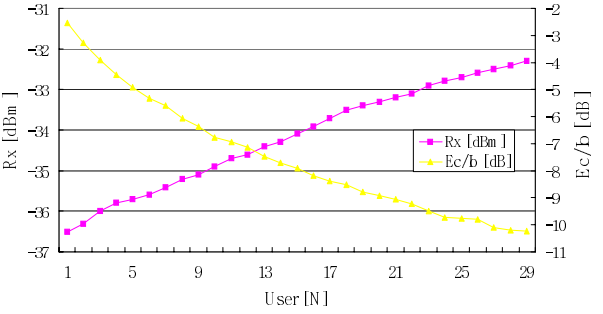


Fig. 4. Ec/Io and Rx by the changes of user

(3) Ec/Io and Rx according to the interference quantity (neighbor base station & several signals)

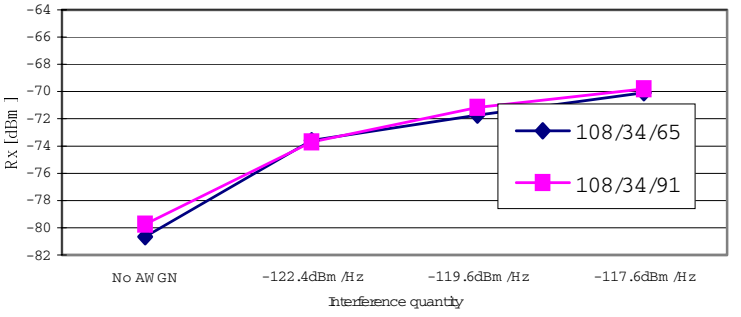
1) Measurements and results

The greatest factor affecting wireless link in the CDMA mobile phone system is the number of user and neighbor base station. Also noise from other communication systems has a great influence and this study conducted the test to measure Ec/Io by the increase in artificial noise as follows. Instead of the influence of other terminal and neighbor base station, AWGN was used to increase interference noise and receiving Rx and Ec/Io was measured by changing D/G of self cell in Fig 5.

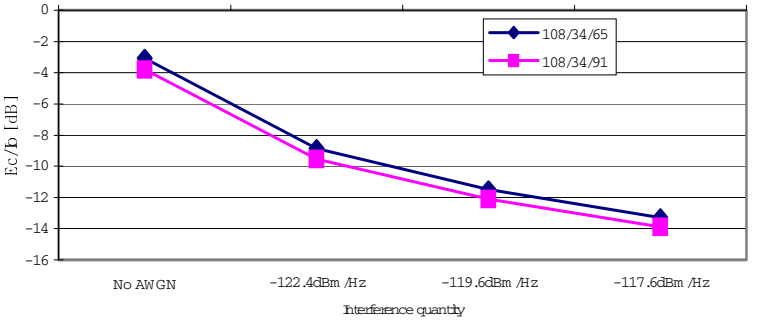
2) Results analysis

As a result of comparing Ec/Io with Rx in artificial noise permission, when noise of -117.6dBm/Hz was permitted, Rx was changed from -80.67dBm to -70.1dBm, suggesting the increase by 10.57dBm. In contrast, Ec/Io deteriorated by 10.2dB from -3.08dB to -13.28dB, suggesting that it had a great influence on noise permission.

While Rx by the increase of D/G of paging channel increased 0.92dBm from –80.67dBm to –79.75dBm without noise and increased 0.28dBm from –70.1dBm to –69.82dBm when the noise of -117.6dBm/Hz was permitted, but Ec/Io deteriorated by 0.62dB in 0.73dB.



(a) Rx



(b) & Ec/Io

Fig. 5. Rx & Ec/Io by the changes in interference quantity

4 Conclusion

In the CDMA cellular mobile communication system, the main factors of deciding good QoS and optimum cell coverage are the strength of field strength and Ec/Io. The filed strength mainly decides the possible area for receiving(coverage) and Ec/Io determines speech quality within decided coverage.

Overhead channels in forward link are assigned output at the adequate rate to optimize the structure of coverage and to minimize the influence of neighbor base station. Besides, the output of pilot channel becomes the standard for deciding the coverage of base station and the quality of forward link. And the quality of forward link depends on the size of received field strength, as well as Ec/Io is the main standard in order to decide the coverage and quality of forward link.

In this study, in order to improve a quality of forward link in the mobile communication system using CDMA, three influencing factors were selected and measured and analyzed through LAB. measurements to identify their effect.

As a result, in order to improve a performance of forward link systems in the mobile communications using CDMA, I confirmed that (1) the increase of power of pilot signal; (2) the proper number of user in a cell; (3) the noise decrease around a cell. These results will contribute to planning the mobile communication networks for the next generation of mobile communications.

References

1. J.Sarnecki, C.Vinodrai, A.Javed, P.O'Kelly, and K.Dick, "Microcell design principles," IEEE Commum, Mag., vol. 31, Apr.1993
2. J.Litva and T.K.Lo, Digital Beamforming in Wireless Communications, Norwood, MA:Artech House, 1996
3. K. Takeo and S. Sato, "Evlauation of CDMA cell design algorithm considering nonuniformity of traffic and base station locations," *IEICE Trans. Fundamentals Electron., Comm. Comput. Sci.*, vol. E81-A, no. 7, pp. 1367-1377, July 1998.
4. T. Rappaport and J.Liberti, *Smart Antennas for CDMA Wireless Systems.* Englewood Cliffs, NJ: Prentice-Hall, 1999.
5. A. Ahmad, "A CDMA Network Architecture Using Optimized Sectoring," *IEEE Trans. Veh. Technol.,* vol.51, pp. 404-410, May 2002.
6. EIA/TIA/IS-95A, "Mobile Station-Base Station Compatibility Standard for Dual-Mode Wideband Spread Spectrum Cellular System," *Telecommunications Industry Association*. July 1993.
7. R. Padovani, "Reverse Link Performance of IS-95 Based Cellular Systems," *IEEE PCS Magazine,* Vol.1, No.3, pp.28-34,1994.
8. L.C. Wang, K. K. Leung, "A high-capacity wireless network by quad-sector cell and interleaved channel assignment", IEEE Journal on selected areas in comm., Vol 18, no. 3, pp.472-480, March. 2000.
9. Dong-You Choi, Sun-Kuk, Noh, "A Study on the Cell Sectorization using the WBTC & NBTC in CDMA Mobile Communication Systems," Lecture Notes in Computer Science, Vol. 3421, pp.920-927, April, 2005
10. Lee, W.C. Y, *Mobile Cellular Telecommunications System.* New York: McGraw-Hill, 1989.
11. A. J. Viterbi, *CDMA, Principles of Spread Spectrum Communication.* New York: Addison-Wesley, 1995
12. A. Fukasawa, T. Sato, Y. Takizawa, T. Kato, M. Kawabe, Reed E. Fisher, Wideband CDMA System for Personal Radio Communications, *IEEE Comm. Magazine,* Oct. 1996, pp. 117-119.
13. A. Jalali and P. Mermelstein, "Power control and diversity for downlink of CDMA systems," in IEEE ICCUPC'93, Ottawa, ON, Canada, pp.980-984, 1993.
14. M. Benthin and K, Kammeyer, "Influence of Channel Estimation on the Performance of a Coherent DS-CDMA System," *IEEE Trans. Veh. Technol.,* vol.46, no.2, pp. 262-267, May 1997.
15. S.H. Min, and K.B. Lee, "Channel Estimation Based on Pilot and Data Traffic Channels for DS/CDMA Systems*," IEEE Global Telecom. Conf.,* pp.1384-1389, 1998.
16. "CDMA System Engineering Training Handbook," Qualcomm Inc., pp.41-48. April 1993
17. An Introduction to CDMA Technology, Qualcomm CDMA Seminar, Oct, 19-20, 1995
18. Samuel C. Yang, CDMA RF System Engineering*, Artech House,* 1998.

Cache Leakage Management for Multi-programming Workloads

Chun-Yang Chen[1], Chia-Lin Yang[2], and Shih-Hao Hung[3]

National Taiwan University,
Taipei, Taiwan 106
r92100@csie.ntu.edu.tw

Abstract. Power consumption is becoming a critical design issue of embedded systems due to the popularity of portable device such as cellular phones and personal digital assistants. Leakage is projected to most amount of cache power budget in 70nm technology. In this paper, we utilize the task-level information to manage cache leakage power. We partition the caches among tasks according to their working set size. We then apply different leakage management policies to the cache regions allocated to active and suspended tasks, respectively. Our proposed policies effectively reduce L1 cache leakage energy by 84% on the average for the multi-programming workloads with only negligible degradations in performances.

1 Introduction

Power consumption is becoming a critical design issue of embedded systems due to the popularity of portable device such as cellular phones and personal digital assistants. In current technology, most of energy consumption is due to switching activities. As the threshold voltage continues to scale and the number of transistors on the chip continues to increase, static energy consumption due to leakage current has become important concern. On-chip caches constitute a major portion of the processor's transistor budget and account for a significant share of leakage. In fact, leakage is projected to account for 70% of the cache power budget in 70nm technology [9] . Therefore, reducing cache leakage power consumption is an important issue for future embedded system design.

Two types of techniques have been proposed to reduce cache leakage: state-destructive and state-preserving. State-destructive techniques use the Gated-Vdd technique to turn off a cache line during its idle cycle [6]. Turning a cache line off saves maximum leakage power, but the loss of state exposes the system to incorrect turn-off decisions. Such decisions can in turn induce significant power and performance overhead by causing additional cache misses that off-chip memories must satisfy. State-preserving techniques use a small supply voltage to retain the data in the memory cell during idle cycles [2]. While state-preserving techniques can only reduce leakage by about a factor of 10 compared to more than a factor of 1000 for destructive techniques, the net difference in power

T. Srikanthan et al. (Eds.): ACSAC 2005, LNCS 3740, pp. 736–749, 2005.

consumed by the two techniques is less than 10% because the state-preserving techniques incurs much less penalty when accessing lower-leakage states.

Previous works on cache leakage management are all based on single-application behavior. In real workloads, caches are actually shared by multiple processes. In this work, we utilize the task-level information to manage cache leakage power. To the best of our knowledge, this paper is the first to study cache leakage control policy using multi-programming workloads. We partition the caches among tasks according to their working set size. We then apply different leakage management polices to the cache regions allocated to active and idle tasks, respectively. They are referred to active-task and idle-task polices in the paper. For the idle-task policy, during a context switch, the cache region allocated to the idle task is turned into low leakage states immediately. We evaluate the performance and power effects of state-preserving and state-destructive techniques under different context switch intervals. For cache regions allocated to the active task, we study three polices. With the state-preserving techniques, a cache is turned into low-leakage mode periodically (i.e, simple policy), or when it is not accessed for a period of time (i.e., no-access policy). The experimental results show that using the state-preserving technique with the simple policy is most effective for active tasks. For the idle task policy, the effectiveness of the state-preserving and state-destructive polices depend on the length of the context switch interval. With higher context switch frequencies, the state-preserving technique performs better than the state-destructive in both energy and performance. That is because with the state-destructive technique, a task suffers from cold cache misses for each context switch. As the time slice increases, the leakage-destructive technique gains it advantage over the state-preserving. The experimental results show that the proposed task-aware cache leakage management scheme can reduce the data cache leakage by 84.3% on the average for multimedia multiprogramming workloads tested in this paper, while the previous proposed approaches which do not utilize the task-level information can only reduce cache leakage by 72.8%.

The rest of this paper is organized as follows. In Section 2, we provide a more detailed view of circuit techniques and control techniques. In Section 3, we detail our leakage control mechanism using cache partition technique. In Section 4, we describe our experimental methodology and present our experimental result. In Section 5, we detail the related works. Finally, In Section 6, we conclude the paper and describe the future work.

2 Background on Cache Leakage Management

The leakage current in cache memories can be managed by circuit techniques and control techniques. A circuit technique reduces leakage by switching off cache lines[12] or putting cache lines into a low-power mode[2]. A circuit technique can be categorized as state-destructive or state-preserving, depending on whether the data in the affected cache lines are lost or not. The reduction of leakage by a circuit technique is highly dependent on the workload that is running on the

system and the control technique that is used to determine when and where a circuit technique is applied. Various control techniques have been proposed, including Dynamically ResIzable (DRI) i-cache[12], simple policy[2], and no-access policy [6].

2.1 Circuit Techniques

State-destructive circuit techniques use ground gating, such as Gated-Vdd[12], which adds one NMOS (n-channel metal-oxide semiconductor) transistor to gate the supply voltage to a memory cell. By switching off unused cache lines, maximum leakage power is saved out of those lines, but the data in those cache lines are lost. The loss of data would incur additional cache misses if the data are to be accessed in the future and would be in the cache if the associated cache lines were not turned off. Since it takes time and power to fetch data from the memory for the additional cache misses, state-destructive circuit techniques may induce significant performance overhead as well as extra dynamic power consumption.

State-preserving circuit techniques reduce leakage current by putting cache lines in a low-leakage state. For example, drowsy caches [2] reduce the level of supply voltage to a cache region when the cache region enters a low-power state (called the drowsy mode), where the information in the cache region is retained but not available for immediate access. To access the cache lines in the drowsy mode, a pseudo miss is generated and a high voltage level (active mode) is required, which incurs one additional cycle overhead to the execution.

Compared to state-destructive techniques, state-preserving techniques take lower performance penalty to access a cache line in low leakage mode as the latency for the cache line to become active is significantly lower than the latency to the next memory hierarchy. On the other hand, while state-preserving techniques may not save as much leakage power as state-destructive techniques, the extra power consumption due to the additional cache misses incurred by a state-destructive technique can offset its leakage power it reduces. Since the additional cache misses also depend heavily on the choice of the control techniques and the application, it is difficult to identify which circuit technique is the winner.

2.2 Leakage Control Policy

A control technique is used to determine when a cache region should enter a low leakage state. The DRI approach[12] resizes the i-cache dynamically based on the cache miss rate during the runtime. When the cache miss rate is low, the cache downsizes by putting part of the cache in a low leakage mode. However, the DRI approach provides no guidance about which part of the cache should be kept active for the near future, thus it can cause excessive pseudo or real cache misses with aggressive downsizing.

The simple policy proposed by Flautner et al. [2] puts all cache lines into a low leakage mode periodically. The policy can be easily implemented without runtime performance monitoring. The policy works best when the selected period matches the rate where the instruction or data working set changes; otherwise, it may perform poorly either because unused cache lines are either kept awake

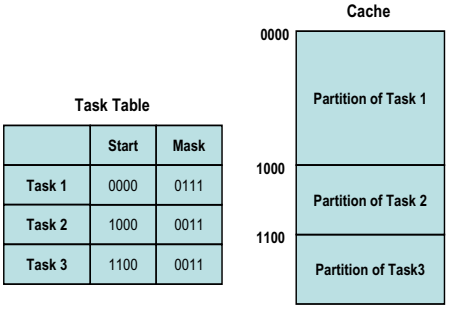


Fig. 1. Task table and its cache mapping. Assume index has 4 bits.

longer than necessary or because useful cache lines are put to the low power mode too early.

The no-access policy proposed by [6] puts a cache line into a low leakage mode when it has not been accessed for a period of time. Compared to the two approaches mentioned above, the no-access policy provides more precise control on individual cache lines and incurs less cache misses. Thus, it should benefit the performance of a state-destructive circuit technique as the overhead caused by an incurred miss is high. The disadvantages of the no-access policy are the cost of the circuits added to monitor and control each cache line and the dynamic and leakage power consumed by the additional circuits.

3 Task-Aware Cache Leakage Management

In a multitasking environment, cache memories are shared among multiple programs. Since cache requirements differ among applications, in this work, we propose to partition cache among tasks according to its working set size. The cache regions allocated to active and idle tasks can use different leakage management polices. The required cache size for each application is obtained through off-line profiling. Since applications for embedded systems are often known a priori, we also perform cache allocation off line. Below we describe the architectural support for cache partitioning and the leakage management policies for the active and idle tasks.

3.1 Architectural Support for Cache Partition

To support cache partitioning, we add a table to the cache keep track of the partition for each task1. Each row in the table contains two fields to keep track of a task: the "start" field contains the starting position of the cache partition that is assigned to the task, and the "mask" field is used to control the maximum size of the cache partition. For the table to map a cache access to a cache line,

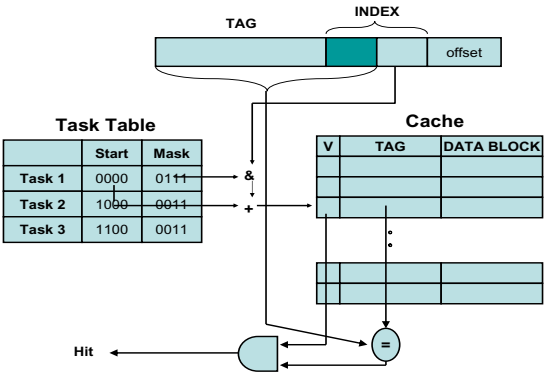


Fig. 2. Task table and address mapping

the start and mask fields both must be N-bit wide, where the number of lines in the cache is 2^N. The mapped cache line can be calculated by:

$start + (index \wedge mask)$

Figure 2 illustrates the mapping scheme. The 4-bit start filed point a task to one of the 16 regions in the cache. "0000" in the start field of Task 1 indicates that the cache partition for Task 1 starts from the region 0. "0111" in the mask field indicates that Task may use cache regions 0 to 7 ("0000" to "0111"). Essentially, the example assigns half of the cache to the task.

Note that as the mask field removes the high-order bit(s) of the cache index, the mapping effectively uses less number of bits as the index to point to a line in a cache partition. Thus, the cache must save more bits in the tag to support the proposed cache partitioning scheme. One should define the smallest partition size (e.g. 2^M cache lines) and add N-M bits to the tag.

When a context switch occurs, we adjust the leakage mode of the cache partitions of the switched tasks using the task table in two steps. The cache leakage management is further discussed in the subsection below.

3.2 Cache Leakage Management Policy

We propose a scheme to manage the leakage for a cache partition based on the state of the task that is assigned to the partition. The state of a task is active if the task is currently running by the processor. The state of a task is idle if the task is not active. Different leakage management techniques are used for active-tasks and idle-tasks.

Active-Task Leakage Management Policy. We consider three options to reduce the leakage for the cache partition of an active task:

1. state-preserving with simple policy.
2. state-preserving with no-access policy.
3. state-destructive with no-access policy.

The simple policy periodically turn off all cache lines, and the no-access policy put a cache line into low leakage mode when it has not been accessed for a time interval. The observing period (i.e.,window size) of both policies are critical for both energy and performance. The details of these policies are described in Section 2.2.

Idle-Task Leakage Management Policy. During a context switch, the cache region allocated to the suspended task is turned into low leakage states immediately. We consider two options to reduce the leakage for cache partition of a idle-task:

1. state-preserving
2. state-destructive

The advantage of the state-preserving policy is that data for the idle-task are still kept in the cache, therefore, it avoids cold start misses during a context switch. Cold start misses incur both energy and performance overheads. However, the state-destructive policy saves more energy during the idle period of a task compared to the state preserving. Therefore, the length of the context switch interval would affect how these two schemes perform. In Section 4.3, we evaluate the energy savings of these two polices with different context switch interval.

4 Experimental Results

In this section, we evaluate the effectiveness of the task-aware cache leakage management scheme. We first analyze the three active task policies with the cache resized to the required cache for each application. After identifying the active-task policy, we then evaluate the idle-task policy with different context switch interval. We then show the energy savings achieved by the task-aware leakage management.

4.1 Experiment Setup

For cache leakage evaluation, we use the HotLeakage toolset [10]. HotLeakage is developed based on the Wattch [1] toolset. HotLeakage explicitly models the effects of temperature, voltage, and parameter variations, and has the ability to recalculate leakage currents dynamically as temperature and voltage change at runtime due to operating conditions, DVS techniques, etc.

To simulate multi-programming workloads, we modified Hotleakage to allow multiple programs executing simultaneously. We implement a round-robin scheduler. Our baseline machine is a single-issue in-order processor. The processor contains a 16KB four-associative D1 data cache. The line size of D1 cache is 32 bytes. The main processor and memory hierarchy parameters are shown in Table 2. Since multimedia applications are the main workloads running on embedded systems, we use applications in the Mediabench [7] and Mibench [3] to

Table 1. Combinations of benchmarks for multiprogramming workload %

	Combination of benchmarks
Workload 1	Mad + Mpeg2decoder + Mpeg2encoder
Workload 2	G721decoder + Mpeg2decoder
Workload 3	Adpcmdecoder + Adpcmencoder + G721decoder + G721encoder

Table 2. Assumed baseline architecture simulation parameters

Processor Core	
Simulator	Hotleakge1.0
Instruction Window	80-RUU, 40-LSQ
Issue width	4 instruction per cycle inorder issue
Functional Unit	4IntALU, 1 IntMult Div
	FPALU, 1FPMult Div
	2 mem ports
Memory Hierarchy	
L1 D-cache	Size 16KB, 4-way LRU, 32B blocks, 1-cycle latency
L1 I-cache	Size 16KB, directly map LRU, 32B blocks, 1-cycle lantency
L2	Unified, 2MB, 2-way LRU, 64B blocks, 12-cycle lantency
Memory	100cycles
Energy Parameter	
Process Technology	0.07um
Supply Voltage	0.9V
Temperature	353

Table 3. Required size for each media benchmark, miss rate constraint equal 1 %

Adpcmdecoder	4K
Adpcmencoder	4K
G721decoder	1K
G721encoder	1K
Mad	8K
Mpeg2decoder	4K
Mpeg2encoder	4K

form multi-programming workloads1. We choose three combinations of benchmarks to construct our multiprogramming workload, see Table 1. To obtain the working set size for each application, we perform off-line profiling. The required cache size is the minimal cache size that has the cache miss rate under 1%. Table 3 lists the required cache size for each application tested in this paper. As for the window size for the control policy, we perform a set of experiments and choose the best window size for the experiment results shown in the section. For the state-preserving technique, the window size for the simple and no-access policies are 2048 and 1024 cycles, respectively. For the state-destructive technique, the The window size for the no-access policy is 4096 cycles.

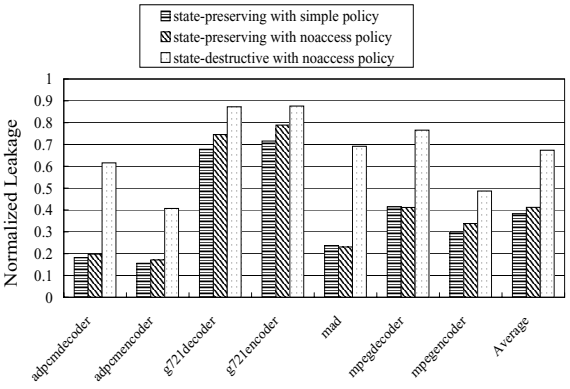


Fig. 3. Normalized L1 data cache leakage of three leakage control policies using in active-task

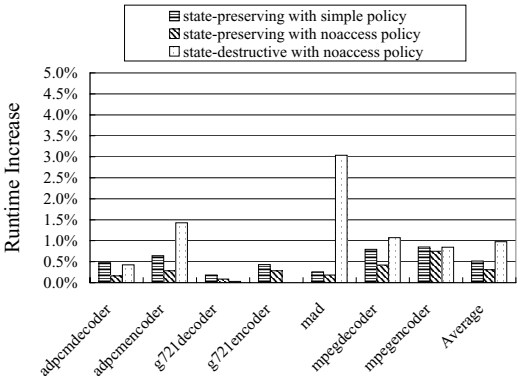


Fig. 4. L1 data cache runtime increasing of three leakage control policies using in active-task

4.2 Active-Task Policy

To evaluate the active-task polices, we first resize the cache to each application's required cache size. We then apply three leakage management policies to the resized cache: state-preserving with simple policy, state-preserving with no-access policy and state-destructive with no-access policy. Figure 3 shows the normalized energy savings over the resized cache without any leakage control mechanism. We can see that the state-preserving technique provides significantly higher energy savings than the state-destructive. This result indicates that if the cache is resized to an application's working set size, there are less opportunities to turn off a line. Furthermore, for the state-preserving technique, the simple policy performs better than the no-access policy because the no-access policy incurs counter overheads. Figure 4 shows the performance degradation caused by

the three active-task policies. In general, the state-destructive technique cause more performance degradation than the state-preserving as expected. As for the state-preserving technique, both simple and nonaccess polices causes less than 1% performance slowdown. Since the non-access policy needs extra hardware resources, for all the experiments below, we use the state-preserving with the simple policy as our active-task leakage management policy.

4.3 Idle-Task Policy

To determine the leakage control policy for idle tasks, we evaluate the state-preserving and state-destructive techniques with different context switch intervals (time slice). The decisions of time slice are based on [5]. Figure5,7,9,6 ,8 and 10 show the normalized energy savings (over the baseline cache without any leakage control) and performance degradations for different time slices. We can see with higher context switch frequencies, the state-preserving technique per-

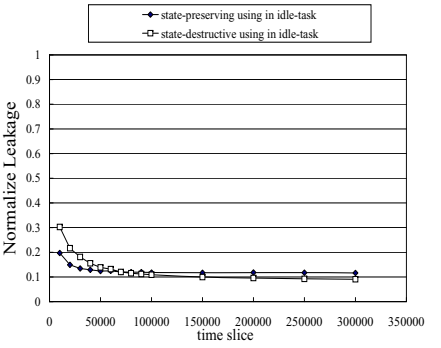


Fig. 5. Workload 1 : Normalized L1 data cache leakage of two leakage control policies using in idle-task

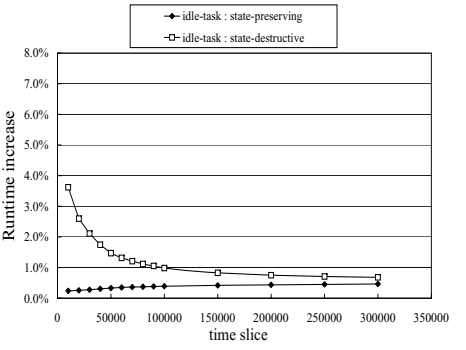


Fig. 6. Workload 1 : L1 data cache runtime increasing of two leakage control policies using in idle-task

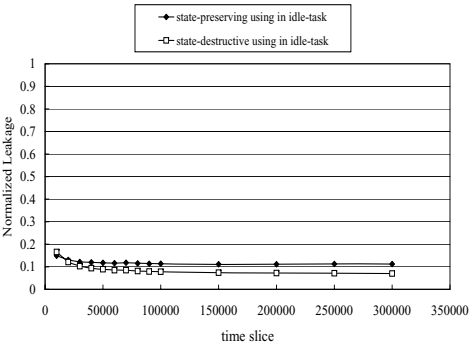


Fig. 7. Workload 2 : Normalized L1 data cache leakage of two leakage control policies using in idle-task

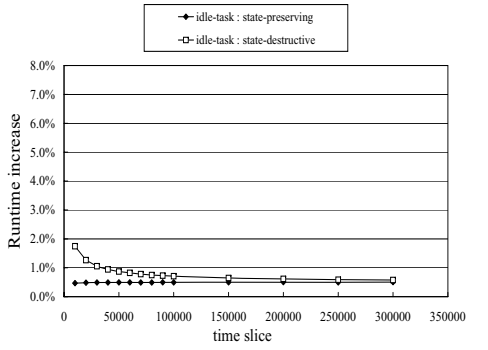


Fig. 8. Workload 2 : L1 data cache runtime increasing of two leakage control policies using in idle-task

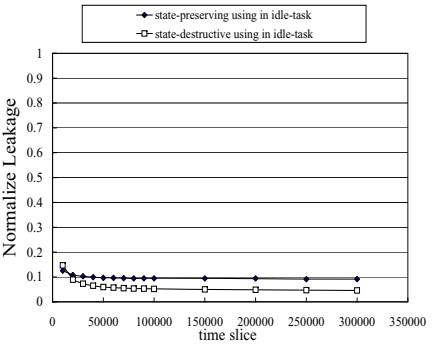


Fig. 9. Workload 3 : Normalized L1 data cache leakage of two leakage control policies using in idle-task

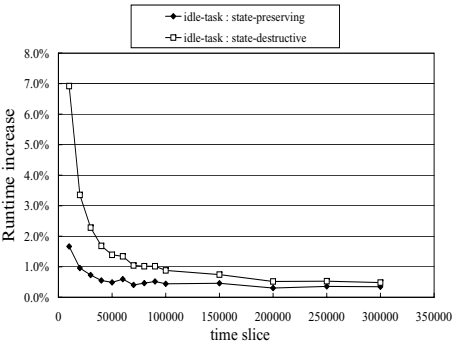


Fig. 10. Workload 3 : L1 data cache runtime increasing of two leakage control policies using in idle-task

forms better than the state destructive in both energy and performance. That is because with the state-destructive technique, a task suffers from compulsory misses for each context switch. Frequent context switch causes significant energy and performance overhead (from accessing the L2 cache). As the time slice increases, the leakage-destructive technique gains it advantage over the state-preserving. The performance degradation of the state-destructive technique also decreases with larger time slice. So for the idle-task policy, the length of the context switch interval is critical for the effectiveness of these two schemes.

4.4 Effects of Task-Aware Cache Leakage Management

In this section, we compare our scheme with three previous proposed methods: state-preserving with the simple policy, state-preserving with the no-access policy and state-destructive with the no-access policy. Note these techniques do

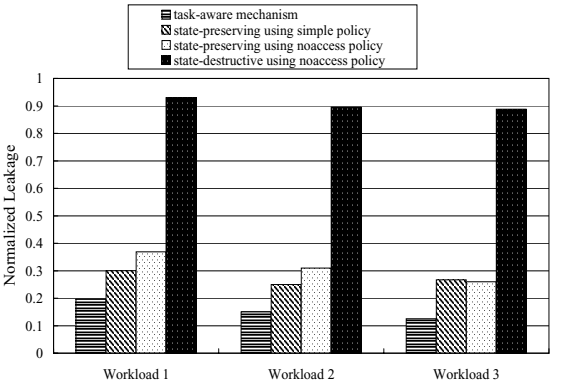


Fig. 11. Comparison of L1 data cache leakage energy

not partition cache among tasks. The experimental results shown in Figure 11 assumes 10000-cycles context switch interval. Our active-task policy is the state-preserving with simple policy. , and our idle-task policy is the state-destructive. The results show that our mechanism achieves most energy savings. The advantage of our task-aware cache leakage management over previous works are two folds. First, we put the cache regions of the suspended tasks into low leakage mode as soon as the context switch occurs rather than waiting for a period of time as in the previously proposed approaches. Secondly, our approach avoids cold miss effects thereby reducing energy consumption from accessing the L2 cache. Note that in Figure **??**, no-access policy is more efficient than simple policy in leakage reduction of Workload 3 by using state-preserving circuit technique. This is because most cache accesses are in a few cache lines in Workload 3, it is efficient to find which cache lines must be preserved in active mode and put into low leakage mode by using no-access policy. Since no-access policy can find the correct working cache lines after occurrence of context switch soon, it is efficient to save leakage at the low context switch interval. The leakage energy reduction provided by our mechanism is 84.3% on the average, while the previous proposed approaches can only reduce cache leakage by 72.8%.

5 Related Work

In the past few years several strategies have been presented for reducing cache leakage energy. DRI I-Cache dynamically resizes and adapts to an application's required size [12]. Using the state-destructive techniques, the cache-decay algorithm [6] (we call it no-access policy in our paper) keeps track of statistics for each cache line and thus may provide better predictive behavior. In [11], the paper argues that adaptive techniques in processor architecture should be desinged using formal feedback-control theory. It use the derivation of a controller for cache decay to illustrate the process of formal feedback-control design and to show the benefits of feedback control.

Drowsy Caches proposed by Flautner et al. [2] provides a state-preserving techniques–drowsy caches–and simple policy, periodically turn off all cache lines, to save the leakage energy without loss the data. Its success depends on how well the selected period reflects the rate at which the instruction or data working set changes. Using the state-preserving techniques, cache sub-bank policy is suggested to use in instruction cache [9]. One mechanism using simple policy is proposed to save more leakage by utilizing the branch target buffer [4].

For save the L2 cache leakage energy, Li.et al [8] takes advantage of duplication in a given cache hierarchy to turn off L2 cache lines when the same data is also available in L1. Using state-preserving leakage control mechanism and state-destructive leakage control mechanism, it investigated five different strategies to put L2 subblocks that hold duplicate copies of L1 blocks in energy saving states.

Different forward hardware-based schemes, some compiler approach are also to save leakage in [14] and [13]. The idea is to keep only a small set of cache lines active at a given time and pre-activate cache lines based on data access pattern.

In [10], by comparing state-preserving and state-destructive at different L2 latencies, it is able to identify a range of operating parameters at which state-destructive is more energy efficient than state-preserving, even though state-destructive does not preserve data in cache lines that have been deactivated. HotLeakage, a simulator of leakage, is proposed in this paper. Its most important features are the explicit inclusion of temperature, voltage, gate leakage, and parameter variations.

6 Conclusions

In the paper, we propose to utilize the task-level information to reduce cache leakage in the multi-tasking environment. We partition the caches among tasks according to their working set size. Cache regions allocated to the active and idle tasks adopt different cache leakage management policy. During a context switch, the cache region allocated to the idle-task is turned into low leakage states immediately. For the active tasks, cache lines are turned into the state-preserving mode periodically. The leakage energy reduction provided by the proposed task-aware cache leakage management is 84.3% on the average for a set of multimedia multiprogramming workloads, while the previous proposed approaches can only reduce cache leakage by 72.8%.

References

1. D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A framework for architectural-level power analysis and optimizations. In *In Proceedings of the 27th International Symposium on Computer Architecture (ISCA)*, 2000.
2. K. Flautner, N. Kim, S. Martin, D. Blaauw, and T. Mudge. Drowsy caches: Simple techniques for reducing leakage power. In *International Symposium on Computer Architecture*, 2002.
3. M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *IEEE 4th Annual Workshop on Workload Characterization*, 2001.
4. J. S. Hu, A. Nadgir, N. Vijaykrishnan, M. J. Irwin, and M. Kandemir. Exploiting program hotspots and code sequentiality for instruction cache leakage management. In *Proc. of the International Symposium on Low Power Electronics and Design (ISLPED'03)*, August 2003.
5. J.C.Mogul and A.Borg. The effect of context switches on cache performance. In *Proceedings of the 4th international conference on architectural support for programming languages and operating systems*, 1991.
6. S. Kaxiras, Z. Hu, and M. Martonosi. Cache decay: Exploiting generational behavior to reduce cache leakage power. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, 2001.
7. C. Lee, M. Potkonjak, and W. H. Mangione-Smith. Media-bench: A tool for evaluating and synthesizing multimedia and communications systems. In *In Proceedings of the 30th Annual International Symposium on MicroArchitecure*, 1997.

8. L. Li, I. Kadayif, Y.-F. Tsai, N. Vijaykrishnan, M. Kandemir, M. J. Irwin, and A. Sivasubramaniam. Leakage energy management in cache hierarchies. In *Proceeding of Eleventh International Conference on Parallel Architectures and Compilation Techniques*, 2002.
9. N. Majikian and T. S. Abdelrahman. Drowsy instruction caches:leakage power reduction using dynamic voltage scaling and cache sub-bank prediction. In *Proc. 35th Ann. Int'l Symp. Microarchitecture (MICRO-35), IEEE CS Press*, pages pp.219–230, 2002.
10. D. Parikh, Y. Zhang, K. Sankaranarayanan, K. Skadron, and M. Stan. Comparison of state-preserving vs. non-state-preserving leakage control in caches. In *In Proceedings of the Second Annual Workshop on Duplicating, Deconstructing, and Debunking in conjunction with ISCA-30*, 2003.
11. S. Velusamy, K. Sankaranarayanan, D. Parikh, T. Abdelzaher, and K. Skadron. Adaptive cache decay using formal feedback control. In *Proc. of the 2002 Workshop on Memory Performance Issues*, May 2002.
12. S. Yang, M. Powell, B. Falsafi, K. Roy, and T.N.Vijaykumar. Gated-vdd: A circuit technique to reduce leakage in deep-submicron cache memories. In *In Proceedings of the 2000 International Symposium on Low Power Electronics and Design*, 2000.
13. W. Zhang, J. S. Hu, V. Degalahal, M. Kandemir, N. Vijaykrishnan, and M. J. Irwin. Compiler-directed instruction cache leakage optimization. In *35th Annual International Symposium on Microarchitecture (MICRO-35)*, November 2002.
14. W. Zhang, M. Karakoy, M. Kandemir, and G. Chen. A compiler approach for reducing data cache energy. In *Proceedings of the 17th annual international conference on Supercomputing*, 2003.

A Memory Bandwidth Effective Cache Store Miss Policy

Hou Rui, Fuxin Zhang, and Weiwu Hu

Institute of Computing Technology, Chinese Academy of Sciences,
100080 Beijing, China
{hourui, fxzhang, hww}@ict.ac.cn

Abstract. Memory bandwidth becomes more and more important in the forthcoming 10 billion transistors chip times. This paper discusses and implements a memory bandwidth effective cache store miss policy. Although the write-allocate policy is adopted, we find it is possible not to load the full cache block from lower memory hierarchy when cache store miss occurs, if the cache block is fully modified before any load instruction accesses the un-modified data of the same cache block. This cache store miss policy will partly reduce the pressure on memory bandwidth, and improve the cache hit rate. We provides a hardware mechanism, Store Merge Buffer, to implement the policy in Goodson-2 processor. Our experiments demonstrate the encouraging results: Memory bandwidth improved by almost 50% (tested by stream benchmark), and IPC on SPEC CPU2K improved by 9.4% on average.

1 Introduction

Memory bandwidth is reported to be a bottleneck in modern processor design [8], especially with the development of On-Chip Multiprocessor(CMP) [15,16] and Simultaneous Multithreaded processor(SMT)[1]. Furthermore, the performance of future microprocessors is highly affected by the limited pin bandwidth. Memory bandwidth is one of the most important questions in the forthcoming 10 billion transistor chip times [8]. Generally, there are two methods to partly overcome the memory bandwidth limitation. One obvious way is to improve the memory interface protocol and increase its frequency. Another way is to find and reduce the avoidable memory traffic. This paper focuses on the latter, and provides a memory bandwidth effective cache store miss policy.

In modern processors, write-allocate caches are normally preferred over non-write-allocate caches. Write-allocate caches fetch blocks upon store misses, while non-write-allocate caches send the written data to lower levels of memory without allocating the corresponding blocks. Comparing these two cases, people found that write-allocate caches lead to better performance by exploiting the temporal locality of recently written data [12]. With the write-allocate policy, however, if one store instruction just modifies part of the cache block (e.g., only one byte is modified), and the cache block does not exist in L1 cache, CPU has to firstly fetch the whole cache block from lower level of memory hierarchy, and then

T. Srikanthan et al. (Eds.): ACSAC 2005, LNCS 3740, pp. 750–760, 2005.

merge the dirty parts with the block fetched back. Compared with non-write-allocate one, this policy generates more memory access requirements. This work investigates the reduction of memory bandwidth requirements of write-allocate caches by avoiding fetches of *fully modified blocks*. A cache block is *fully modified* if some adjacent stores modify the whole cache block before any subsequent load instruction accesses the un-modified data of the same cache block. Hence, the fetches of *fully modified blocks* can be avoided without affecting program correctness.

This work has three contributions: 1) We explore the potential to effectively reduce both memory traffic and cache misses by directly installing *fully modified blocks* in data caches, thus improve the efficient memory bandwidth and the performance. 2) We introduce a hardware mechanism, Store Merge Buffer(SMB), which can efficiently identify avoidable misses by delaying fetches original blocks for store misses and merge them. The SMB has certain advantages over schemes such as write-validate caches [12] and cache installation instructions [11], as well as the Store Fill Buffer(SFB) [14]. 3) We implement this cache store miss policy in Goodson-2 processor [3], which results in 9.4% performance speedup across SPEC CPU2000 benchmarks on average, and improves the memory bandwidth by 50%.

The rest of the paper is organized as follows: Section 2 discusses related works, and Section 3 makes a simple introduction of the Goodson-2 Processor, describes the simulation environment and evaluation methodology. Section 4 has an analysis on the cache store miss, and characterizes the avoidable memory traffic. Section 5 proposes the Store Merge Buffer and evaluates its performance impact. Finally, we conclude the work and make an acknowledgement in Section 6 and 7.

2 Related Works

There have been many studies on reducing memory traffic. The write-validate cache [12], in which store allocated blocks are not fetched, is one of such schemes. In write-validate cache, the data is written directly into the cache, and extra valid bits are required to indicate the valid (i.e. modified) portion of the blocks. One of write-validate's deficiencies is the significant implementation overhead, especially when per-byte valid bits are required (e.g. MIPS ISA [6]). More importantly, a write-validate cache reduces store misses at the expense of increased load misses arising from reading invalid portions of directly installed blocks, which may negate write-validate's traffic advantage. As a comparison, the Store Merge Buffer reduces both load and store misses, and incurs far less overhead to yield comparable cache performance to a write-validate cache.

Cache installation instructions, such as dcbz in PowerPC [11] and the cache instruction of creating dirty executive data in MIPS ISA [6], are proposed to allocate and initialize cache blocks directly [10]. Unfortunately, several limitations prevent broader application of the approach. For example, to use the instruction, the compiler must assume a cache block size and ensure that the whole block

will be modified. Consequently, executing the program on a machine with wider cache blocks may cause errors. The use of the instruction is further limited by the compiler's limited scope since it cannot identify all memory initialization instructions.

Store Fill Buffer(SFB) [14] is similar to our work. By delaying fetches for missed stores, SFB identifies the majority of fully modified blocks even with a size as small as 16 entries. However, there are important differences from our works. SFB is parallel with the L1 cache, and then the index is likely to be virtual address because the virtual address indexed L1 cache is normally preferred in modern CPU. In contrast, The Store Merge Buffer is placed in the memory interface of CPU chip, what's more, Store Merge Buffer is indexed by physical address. The policy of virtual address index usually can be affected by the operating system, e.g. it must save and restore the context of SFB while the Operating System switches the context. Our work can avoid this. And the performance of SFB is estimated for Alpha ISA simulator[7], while our work discusses and implements the cache store miss policy in our full-system MIPS ISA simulator.

3 Methodology

This section firstly make an simple introduction about Goodson-2[1] processor. Then the simulator and the benchmark are described.

3.1 The Goodson-2 Processor

The Goodson project [2,3] is the first attempt to design high performance general-purpose microprocessors in China. The Goodson-2 processor is a 64-bit, 4-issue, out-of-order execution RISC processor that implements a 64-bit MIPS-like instruction set. The adoption of the aggressive out-of-order execution techniques (such as register mapping, branch prediction, and dynamic scheduling) and cache techniques (such as non-blocking cache, load speculation, dynamic memory disambiguation) helps the Goodson-2 processor to achieve high performance even at not so high frequency. The Goodson-2 processor has been physically implemented on a 6-metal 0.18 um CMOS technology based on the automatic placing and routing flow with the help of some crafted library cells and macros. The area of the chip is 6,700 micrometers by 6,200 micrometers and the clock cycle at typical corner is 2.3ns.

Figure 1 shows the memory hierarchy. This CPU has 64KB Data cache and 64KB instruction cache, and has off-chip 8MB L2 cache. And the L1 data cache is write-allocate, and write-back. In order to parallel the TLB and L1 cache access, the L1 cache is indexed by virtual address. There are two queues in the cache interface, which we use the SysAD protocol. More details are included in paper[3]. The next generation of Goodson-2 will integrate on-chip L2 cache and DDR memory controller.

[1] Goodson-2 is also named godson-2 in [2,3]. Now, we use the name "goodson-2".

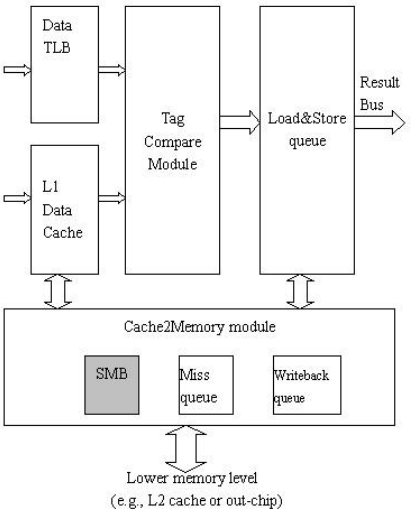


Fig. 1. The memory hierarchy of Goodson-2 processor(the gray block is new part added in our work)

Table 1. Goodson-2 Processor Parameters

Parameter	Value
Functional Units	3 FP, 6 Int
Pipeline depth	9 stages
Load & Store queue	16 entries
Instruction Windows Size	32-entry FP, 32-entry Int
Instruction Cache	64kB, 2-way, 64 byte/line
Data Cache	64kB, 2-way, 64 byte/line
L1 cache hit latency	3 cycle
L2 Cache	no(off-chip)
I/DTLB	48/128 entry
Latency(to CPU)	L2 6 cycle, Mem 80 cycles
branch predictor	Gshare, RAS, BTB

3.2 Cycle-by-Cycle Simulator

We developed our own full-system cycle-by-cycle simulator, which is used to build the processor prototype and make performance analysis. Table 1 shows the detail configuration. Our experiments show that the simulator can match the real CPU chip quite well, the error range is within 5%. The configuration parameters are the same as Table 1, except for ignoring the simulation of L2 off-chip cache.

3.3 Benchmark

To perform our evaluation, we run SPEC CPU2000 benchmarks [13] on our simulator except for the 178.galgel, 187.facerec, 189.lucas, due to our compiler reason. The benchmarks are compiled with the peak settings, which perform many aggressive optimizations. The train data set is chose due to the long time simulation. And we run the STREAM memory bandwidth benchmark [5] on our simulator, which is an industry-standard benchmark suite that does an excellent job of measuring sustainable memory bandwidth. All the benchmarks run on our full-system simulator.

4 Analysis of Cache Store Miss

Cache miss may have great effects on performance. When a load instruction does not hit the cache, a cache load miss will occur. The load must wait for the cache controller to fetch the corresponding cache block from lower levels of memory. And those following instructions, which will consume the load result, must wait in the issue queue or reservation station, and then the pipeline may be stalled. On the contrary, cache store miss cannot lead to the delay of other instructions. Usually the architects consider many kinds of technologies to hide the cost of delinquent load miss, such as the prefetch, pre-execution, as well as Out-of-Order execution with larger instruction window and so on, and ignore the cache store miss. However, the cache store miss also has great important impacts on performance. The reason is mainly due to the memory bandwidth requirement caused by the cache store miss. Modern CPUs adopt the write-allocate policy. As we know, usually the size of cache block is larger than the content of the store instruction. When a cache store misses, CPU must fetch the whole cache block from the lower memory hierarchy. The store will not commit its content until the clean cache block is fetch back into the L1 cache. In a word, the store instruction may put pressure on memory bandwidth. The more frequent the store miss is, the heavier the memory traffic is. The busy

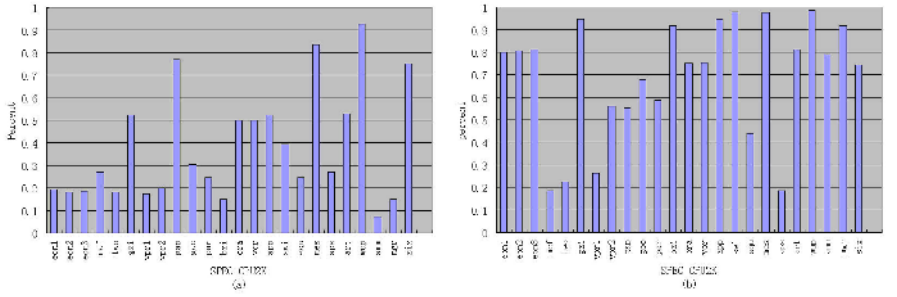


Fig. 2. (a) The proportions of store miss vs. whole cache miss; (b) The proportions of *fully modified* cache store miss vs. total cache store miss

memory hierarchy will delay other delinquent load instructions, and then the whole performance will decrease. Fortunately, when the fully modified blocks are frequent, avoiding fetching the fully modified blocks will greatly reduce the pressure of the memory bandwidth requirement. It is worth improving the cache store miss policy.

Though analyzing the program behavior, we characterize the avoidable memory traffic due to the fully modified blocks. Figure 2(b) demonstrates that, for SPEC CPU2000, the fully modified store miss is 69.21% of the whole cache store miss. That indicates that a large number of cache store miss do NOT need to fetch the original cache block from the lower memory hierarchy, especially form the off-chip memory or cache. And Figure 2(a) shows that store miss also amounts a large part of the whole cache miss, on average 37.8%. According to these statistics, improving the cache store policy has great performance potential. Hence, the fetches of silent fully dirty blocks can be avoided without affecting program correctness with the Store Merge Buffer promoted in this paper. This work investigates the reduction of memory bandwidth requirements of write-allocate caches by avoiding fetches the fully modified blocks in case of cache store miss.

5 Store Merge Buffer

5.1 The Design of Store Merge Buffer

We promote Store Merge Buffer (SMB) to avoid the fetching of *fully modified blocks*. The SMB is a content addressed memory (CAM) in essence.

In case of L1 cache store miss, the miss request (including the miss address and the store mask, as well as the store content) will enter in the SMB. And the SMB will find whether there is a match in its current addresses. If some entry matches, then the request will be merged in this entry according to its mask, at the same time the modified content and mask are updated. If not, a new entry will be allocated. If the SMB is full, it should not accept the miss request(The miss is stall in load-store queue).The oldest request in SMB are sent to the lower memory hierarchy, and the return result will be merged with the current content in the SMB entry. If a fully modified cache block is found, it will be directly installed in the L1 cache by refill bus, and then avoid fetching the cache block from lower memory hierarchy. In this way, the memory bandwidth requirement are reduced.

Furthermore, when one subsequent load does not hit the L1 cache, it will query the SMB, if hit, it will return the result directly. In this way, the cache hit rate is improved.

There are two important questions in the design of SMB.

1) Where does SMB lie in? One Obvious choice is that, the SMB is accessed in parallel with the L1 cache (just like the SFB). However, modern processors are usually indexed by virtual address, which lead to the unknown physical address when accessing the L1 cache. So the SMB has to be indexed by virtual address. When the Operating system switches the running processes or tasks, the SMB

has to be stored as one of the process context and to be restored when the process resumes, because different processes may have the same virtual address and the different physical address. This will increase the cost of process switch and affect the SMB's identification of *fully modified blocks*. SFB just does this way. If we index the SMB with the request's physical address, SMB will be transparent to process switch. Then we put the SMB somewhere the physical address is known, e.g. Load/Store queue, or the memory interface module. We will consider this question again in section 5.2.

2) How large is SMB ? The size of SMB decides the instructions range SMB can observe. That is to say, the larger the window is, the more cache store miss instructions SMB can merge. SMB make an analysis on the interval distance of fully modified store. The distance is the number of data references executed during the period that the whole block is overwritten. A block with long fill intervals has a higher probability to be partially modified in case that its lifetime is short. Hence, the lengths of fill intervals reveal the stability of fully modified blocks. On the other hand, large size of SMB are not practical in physical design. Our experiments show the encouraging result that SMB with 4 entries is enough.

5.2 Implementation Details

In this section, we discuss the SMB implementation details in Goodson-2 simulator. According to the above discussions, we implement the SMB in the memory interface module. The SMB is indexed by physical address. Different configurations will be experimented on the cycle-by-cycle execution-driven full system simulator.

First of all, we introduce the details of the memory interface module and the corresponding modules. In our design, the memory interface module is named cache2memory module, and the load/store queue is put behind the L1 data cache module (Figure 1).

There are two queues in cache2memory module, which are miss request queue and write back queue. All miss requests from the load/store queue will reside in the miss request queue, and all dirty cache blocks replaced by the cache conflict or returned by some cache instruction, will reside in the write back queue. In each cycle, a request in the miss queue is sent out to the memory bus if the bus is not busy. If the bus is free and there are no other miss requests in miss request queue, one dirty cache block in the write back queue is sent back to the lower memory hierarchy. When a new miss request enters the cache2mem, it will first look up any match in write back queue. If exists, directly return the block. if not, it will look up the miss queue. If a match is find, this request will be ignored. If no match, a new entry will be allocated.

When SMB is added in the cache2memory module, some mechanisms are needed in order to maintain the data coherence. That is to say, CPU need merge and fetch the latest version of data at any time. Firstly, a cache store miss will look up the miss queue, if match, then it just resides in the miss queue, for the entry in miss queue will soon be sent out and the return block should be merged with the store content according to its mask. If no match, the cache store miss request will enter the SMB, and also try to match each entry in SMB. If some entry matches,

the request should be merged in the entry according to its mask, and then the dirty content and mask are updated. If not, a new entry will be allocated. In order to make little modifications, we make use of the original mechanisms as much as possible. When SMB is full or SMB finds a fully modified cache block, it will directly moved this entry to the miss request queue. According to the mask, the miss request decide whether to directly install the fully modified blocks into the L1 cache by refill bus, or to sent the miss request to the lower memory hierarchy.

5.3 Results

We configure 16 entries of SMB. Firstly, we run the STREAM memory bandwidth benchmark [5] on our simulator, the result is listed in Figure 3. Figure 3(a) shows the result of the base configuration with SMB, and the Figure 3(b) gives the result of the base configuration without SMB. Obviously, the memory bandwidth is significantly improved, from 81MB/s to 127MB/s, improved by 56%. The main reason is that almost all store misses are fully modified store miss, and they are so compact that these instructions can all be merged in SMB, without the need to go out of chip. Thus, the bandwidth will be greatly improved. By the way, the memory bandwidth is lower than other processor, mainly because of the lower frequency (the frequency of front bus is 100MHz, and the on chip frequency is 300MHz) configured in our simulator.

Then we run SPEC CPU2000 on our simulator. Figure 5 lists the result. For all the programs, SMB does not have detrimental effect. And for gzi, app, SMB improves the IPC by over 50% percent. The reason is that there are many adjacent cache store misses, which access the sequent memory data. And from Figure 2, we can see that the cache store miss is more than 50 percent of the whole cache miss for these two programs. So, the store buffer can identify and merge most of the store cache miss, which will effectively improve the performance. On average, the performance for all the SPEC CPU2000 is improve by 9.4

Function	Rate (MB/s)	RMS time	Min time	Max time
Copy:	127.1302	0.0255	0.0252	0.026
Scale:	112.3909	0.0285	0.0285	0.0286
Add:	115.1494	0.0417	0.0417	0.0417
Triad:	111.7057	0.043	0.043	0.043

(a)

Function	Rate (MB/s)	RMS time	Min time	Max time
Copy:	81.6723	0.0392	0.0392	0.0393
Scale:	81.3858	0.0394	0.0393	0.0394
Add:	86.1838	0.0557	0.0557	0.0558
Triad:	86.5489	0.0555	0.0555	0.0555

(b)

Fig. 3. Stream benchmark result

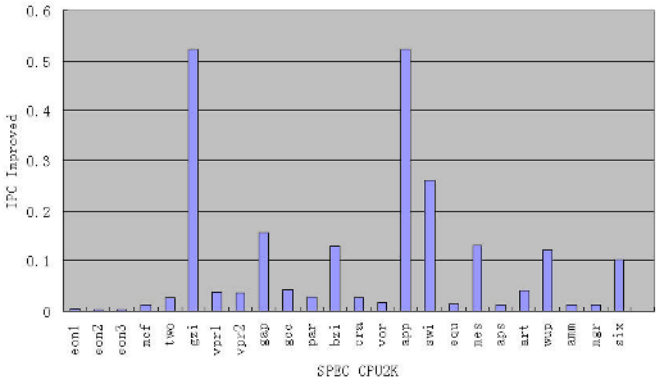


Fig. 4. Performance improved with the optimization of 16 entries SMB for SPEC CPU2000

In conclusion, SMB have three advantages:

1) Avoid the *fully modified* store misses SMB identifies , and thus reduce the unnecessary bandwidth requirement;

2) Promote the execution speed of store instructions, and have an active impact on the utilization of load/store queue and reorder buffer queue;

3) For cache store miss, we delay and merge all the possible corresponding miss requests. This may increase the lifetime of other cache block now in the L1 cache, which may increase the load hit rate. That is to say, SMB will decrease the cache miss rate;

Figure 3 and 4 show that SMB has great performance potential. However, when moving our design from the simulator to the physical design, more things need to be considered. First, we make an evaluation about the window size of SMB. When the size is configured as 1, 4, 8, 16, 128 and so on, the result show that, SMB with 4 to 16 entries have generated almost the same performance. From section 5.2, we place SMB in cache2mem module. Hence, there are three queues in cache2mem, and each queue need to look though the other two before it is going to insert a new entry. This may add the time needed by cache2mem module, and thus may have impact on the whole cycle time. Fortunately, the cache2mem module is out of the main pipeline, so the added time has little effect on the performance. Furthermore, some alternatives maybe help overcome this problem. We merge the SMB with miss queue or load/store queue. This will decrease the number of queue, and avoid unnecessary inter-queue looking up. Our colleagues are implementing the policy on the next version of goodson-2 processor.

6 Conclusion

Memory bandwidth limitation will be one of the major impediments to future microprocessors. Hence, reducing memory bandwidth requirements can improve

performance by reducing pressure on store queues and cache hierarchies. This paper focuses on the reduction of memory bandwidth requirements of write-allocate caches by avoiding the fetching of *fully modified blocks* from lower memory hierarchy. A cache block is *fully modified* if some adjacent stores modify the whole cache block before any subsequent load instruction accesses the un-modified data of the same cache block. Hence, the fetching of *fully modified blocks* can be avoided without affecting program correctness.

We propose a memory bandwidth effective cache store miss policy, and implement a hardware mechanism, Store Merge Buffer, to identify *fully modified blocks* and thus reduce memory traffic. By delaying fetching for store misses, the Store Merge Buffer identifies the majority of *fully modified blocks* even with a buffer size as small as 4 entries. Moreover, Store Merge Buffer reduces both load and store misses. We implement the policy in Goodson-2 processor simulator, our experiments demonstrate the encouraging result: Memory bandwidth improved by almost 50% (tested by STREAM benchmark), and IPC on SPEC CPU2000 improved by 9.4% on average. Our colleagues are adding the RTL codes about the Store Merge Buffer in the next version of Goodson-2.

This memory bandwidth effective cache store miss policy can be applied not only in super scalar processor, also in the CMP and SMT processors, in which the memory bandwidth limitations will be more serious. Our future work will exploit the potential of this policy on CMP and SMT processors.

Acknowledgements

We would appreciate the anonymous reviewers for their advices. This work is supported by the National Natural Foundation of China for Distinguished Young Scholars under Grant No.60325205; the Basic Research Foundation of the Institute of Computing Technology, Chinese Academy of Sciences under Grant No.20056020; Knowledge Innovation Project of the Institute of Computing Technology, Chinese Academy of Sciences under Grant No.20056240; the National High-Tech Research and Development Plan of China under Grant No.2002AA110010, No.2005AA110010; Knowledge Innovation Program of Chinese Academy of Sciences under Grant No.KGCX2-109.

References

1. D. M. Tullsen, S. J. Eggers, H. M. Levy et al, "Simultaneous Multithreading: Maximizing On-Chip Parallelism", *22nd Annual International Symposium on Computer Architecture*, 1995.
2. Weiwu Hu, Zhimin Tang, "Microarchitecture design of the Godson-1 processor", *Chinese Journal of Computers*, April 2003, pp.385–396. (in Chinese)
3. Wei-Wu Hu, Fu-Xin Zhang, Zu-Song Li, "Microarchitecture of the Godson-2 Processor", *Journal of Computer Science and Technology*, Vol.20, No.2, March 2005.
4. Divid Patterson, John Hennessy, "Computer Architecture: A Quantitative Approach", *Morgan Kaufmann Publishers, Inc.*, 1996.

5. John D. McCalpin, "STREAM: Sustainable Memory Bandwidth in High Performance Computers", *http://www.cs.virginia.edu/stream/*
6. Kenneth Yeager, "The MIPS R10000 superscalar microprocessor", *IEEE Micro*, April 1996, 16: 28–41.
7. Kessler R, "The Alpha 21264 microprocessor", *IEEE Micro*, March/April 1999, 19: 24–36.
8. Doug Burger, James R. Goodman , Alain Kagi, "Memory Bandwidth Limitations of Future Microprocessors", *ISCA*, 78-89, 1996.
9. Tien-Fu Chen , Jean-Loup Baer, "A performance study of software and hardware data prefetching schemes", *the 21st Annual International Symposium on Computer Architecture*, pages 223–232, 1994.
10. W. Wulf , S. McKee, "Hitting the Memory Wall: Implications of the Obvious", *ACM Computer Architecture News*, V.23, No.1, 1995, pp. 20-24.
11. IBM Microelectronics and Motorola Corporation, "PowerPC Microprocessor Family: The Programming Environments,Motorola Inc., 1994.
12. N. Jouppi, "Cache Write Policies and Performance", *ACM SIGARCH Computer Architecture News,* ,V.21, No.2, May 1993, pp. 191-201.
13. J. L. Henning, "SPEC CPU 2000: Measuring CPU Performance in the new millennium", *IEEE Computer*, July 2000.
14. Shiwen Hu, Lizy John, "Avoiding Store Misses to Fully Modified Cache Blocks", *Submitted to EURO-PAR 2005*, October 2005.
15. J. Huh, D. Burger, S. Keckler, "Exploring the design space of future CMPs", *In The 10th International Conference on Parallel Architectures and Compilation Techniques*, pages 199–210, September 2001.
16. Doug Burger, James R. Goodman, "Billion-transistor architectures: there and back again ", *Computer*, Volume 37, Issue 3, Mar 2004 Page(s):22 - 28.

Application-Specific Hardware-Driven Prefetching to Improve Data Cache Performance*

Mehdi Modarressi, Maziar Goudarzi, and Shaahin Hessabi

Computer Engineering Department, Sharif University of Technology, Tehran, Iran
{modarressi, gudarzi}@mehr.sharif.edu, hessabi@sharif.edu

Abstract. Data cache hit ratio has a major impact on execution performance of programs by effectively reducing average data access time. Prefetching mechanisms improve this ratio by fetching data items that shall soon be required by the running program. Software-driven prefetching enables application-specific policies and potentially provides better results in return for some instruction overhead, whereas hardware-driven prefetching gives little overhead, however general-purpose processors cannot adapt to the specific needs of the running application. In the application-specific processors that we develop customized to an object-oriented application, we implement application-specific hardware prefetching to benefit from both worlds. This prefetching policy prefetches all data items that shall be unconditionally accessed by a class method when the class method is called. We mathematically analyze this mechanism and present its simulation results using some object-oriented benchmarks. Simulation results in absence and presence of the proposed prefetching mechanism confirm the theoretical results and show that on average, the miss ratio is reduced by 73%.

1 Introduction

Using cache memories is a way for bridging the performance gap between the high-speed processors and low-speed memory chips. Up to now, many methods such as prefetching have been proposed to improve the cache behavior [1].

Data prefetching brings up data items in the memory hierarchy (here, in cache memory) before the actual request for them. Prefetching increases processors performance by improving the cache hit ratio. Prefetching has been implemented in almost all general purpose processors, often by fetching multiple-words as a cache block to take advantage of the spatial locality of references. This prefetching scheme belongs to a class of data prefetching methods, called hardware-driven methods.

In hardware-driven methods, the prefetching algorithm is implemented in hardware. These methods can prefetch the data items independently and have no overhead for the processor. On the other hand, hardware-driven methods cannot adapt the running application and most of them are bound to take advantage of the spatial locality using a constant, and often simple, algorithm.

* This work is supported by a research grant from the Department of High-Tech. Industries, Ministry of Industries and Mines of the Islamic Republic of Iran.

T. Srikanthan et al. (Eds.): ACSAC 2005, LNCS 3740, pp. 761 – 774, 2005.

To adapt the prefetching mechanism to the running application, another class of prefetching methods, compiler-driven or software prefetching mechanisms, is proposed. In software-driven methods, the data items that will be needed in the near future would be requested by the running program. These methods require some degree of hardware support. Since the prefetched data items are determined by the program, software methods are completely application specific and should give the highest hit ratio.

But this class of prefetching mechanisms causes the execution of some additional instructions to calculate the addresses and request the prefetched data. This instruction overhead increases the execution time of the program and even may exceed the impact of hit ratio improvement. In the next section, we will present a survey on some data prefetching mechanisms.

In embedded systems, where the system structure (hardware and software) and the scope of the system application is known to the designer and remains reasonably unchanged during the system life, some system details and properties can be used to improve the performance.

The application specific processor we use in this research is introduced in Section 3. This processor is synthesized from an object-oriented high–level specification. Here, we have used one of the attributes of the system (object fields on which every class method works) for data prefetching and improving the data cache hit ratio. In this scheme, the cache controller prefetches all fields of an object required by a class method, when the class method is invoked. This approach adapts the prefetching mechanism to the application-specific processor and, therefore, to the running application. Thus, the proposed method can potentially have a hit ratio improvement near the software-driven methods. On the other hand, this method is implemented in hardware and does not have the overheads of software-driven methods.

In this paper, after proposing the prefetching method, we present the mathematical analysis and its simulation results on some object-oriented benchmarks.

The structure of the paper is as follows. In Section 2, we review some of the prefetching mechanisms. In Section 3, we introduce our embedded system architecture and propose our prefetching mechanism. Section 4 contains the mathematical analysis of the proposed method. In Section 5, we present the experimental environment. Section 6 contains the simulation results, and finally Section 7 concludes this paper.

2 Data Prefetching Mechanisms

Data prefetching is one way to masking and reducing the memory latency in processors. Many prefetching techniques including hardware, software, or integrated hardware-software techniques have been proposed which increase the cache hit ratio by adding prefetching capabilities to a processor.

The main idea in hardware prefetching mechanisms is to take advantage of the spatial locality of memory references through prefetching the memory blocks near (or in regular distance from) the current fetched block. Using this approach, these techniques offer their best performance in loops working on large arrays.

The most common hardware prefetching technique is the use of multiple-word cache blocks. In this technique, by grouping consecutive memory words into a single unit, caches exploit the principle of spatial locality to implicitly prefetch data that is likely to be referenced in near future. However, increasing the cache block size may cause some problems since the cache controller operates only on whole cache blocks; for example in the case of changing only a word in a block the entire block should be considered as a modified block, which increases the bus load of write back or cache consistency mechanisms.

Sequential prefetching is a subclass of hardware prefetching techniques based on spatial locality. The simplest sequential prefetching scheme is the *one block lookahead (OBL)* approach. This method initiates a prefetch for block *b+1* when block *b* is accessed. A form of this method is used in HP PA7200 processor [2].

It is also possible to prefetch more than one block after a demand fetch to some value greater than one. But, this increases the bus traffic and usually prefetches many unused blocks (wrong prefetchings) [3].

An *adaptive sequential prefetching* policy has been proposed in [4] that allows the value of prefetched blocks to vary during program execution by periodically calculating the current spatial locality characteristics of the program. Sequential prefetching methods can be implemented with relatively simple hardware. However, these methods perform poorly when non-sequential memory access patterns are encountered.

Several techniques have been developed which monitor the processor's address referencing pattern by comparing successive addresses used by load or store instructions to detect constant stride array references originating from loop structures. In one method proposed in this category, from the difference of two successive addresses of a memory instruction, the next reference address of the instruction may be predicted [5]. This approach requires the previous address used by a memory instruction and last detected stride to be stored. Since recording the reference history of each memory instruction in program is clearly impossible, a table called the *reference prediction table* holds the history of the most recently used memory instructions. In order to predict irregular reference patterns, a Markov predictor for data prefetching has been proposed in [6]. In this method, using a hardware table and dynamically registering sequences of cache miss references, the prefetcher predicts when a previous pattern of misses has begun to repeat itself. When the current cache miss address is found in the table, prefetches for likely subsequent misses are issued to a prefetch request queue.

Another approach to data prefetching is software (or compiler-driven) prefetching. The main idea of software prefetching is to insert prefetch instructions in the program to request the data before they are needed. The prefetch instructions have been implemented in some processors such as *hp PA8000* and *MIPS R10000* [1].

Although implementation of prefetch instructions will vary, all prefetch instructions are invisible to a program; they do not cause an exception for page faults and protection violations and do not change the memory or register contents. Prefetch instructions are useful only if they do not interrupt the work of processor. To do this, Prefetch instructions should be implemented as non-blocking memory operations to allow the cache to continue to supply data while waiting for prefetched data.

Inserting the prefetch instructions in the program may be done by compiler or by hand. However, the use of prefetch instructions adds processor overhead because they require extra execution cycles, and fetch source addresses calculation. Therefore, to ensure that the overhead does not exceed the benefits, we should use prefetch instructions precisely and only prefetch the references that are likely to be cache misses. This prefetching technique is most often used within loops in array calculations which in many cases have predictable array referencing patterns. Some techniques, like software pipelining and loop unrolling, can increase the performance of the software prefetching methods by overlapping execution with the data prefetching [1], [7]. The hardware required to implement software prefetching is less than hardware prefetching techniques.

The last approach to data prefetching is integrating hardware and software prefetching. The majority of these prefetching schemes are concentrated on using hardware prefetching while supplying its parameters in the compile time. For example, a programmable *prefetch engine* has been proposed in [8] as an extension to the reference prediction table described before. In this prefetching engine, address and stride information are supplied by the program rather than being dynamically established in hardware.

A method for prefetching objects into an object cache is suggested in [9]. This method uses references within the first object to prefetch a second object into a cache memory and presents a structure for an object cache.

Software prefetching relies exclusively on compile-time analysis to insert fetch instructions within the program. Although these techniques offer the best hit ratio, they suffer from instruction overhead and lack of run-time information. On the other hand, the hardware techniques perform prefetching based on run-time information and without any compiler or processor support. Thus, hardware-driven methods cannot adapt to running application and only employ the spatial locality of references to improve the cache behavior. In this paper, we present an application-specific hardware prefetching technique, in application-specific processors that we develop customised to an object-oriented application. This method adapts to the running software while is implemented in hardware and therefore can give a hit ratio near the software-driven methods with no performance overhead.

3 Application-Specific Data Prefetching in Hardware

3. 1 Embedded System Architecture

The embedded system architecture that we follow in this research is depicted in Fig.1. The system is a Network-on-Chip (NoC) architecture that consists of a processor core along with a set of hardware functional units (FU). The architecture is specifically designed to suit object-oriented (OO) applications. A typical OO application defines a library of classes, instantiates objects of those classes, and invokes methods of those objects. Our implementation approach for each of these three major components of an OO application is described below. For presentational purposes, we follow the C++ syntax in describing each component.

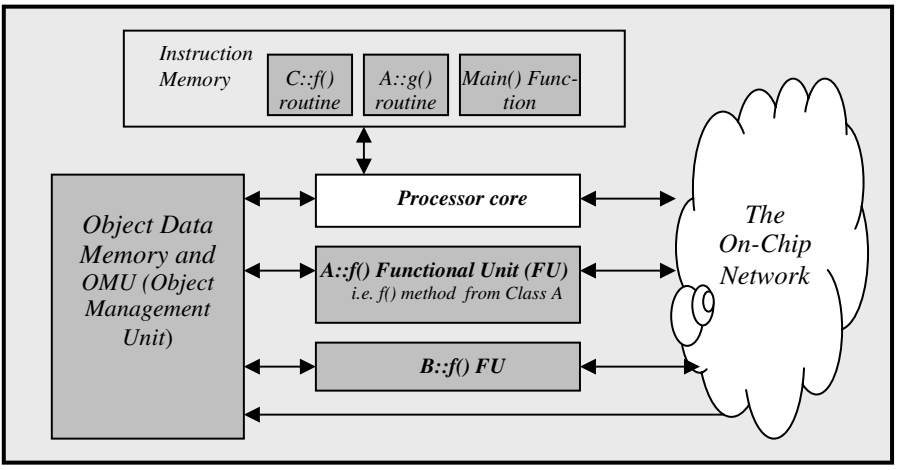


Fig. 1. The internal architecture of an OO-ASIP

- **Class library:** Each class consists of variable declarations and method definitions. Variable declarations are compile-time information and do not require a corresponding component in implementation. Methods of the OO class library are either implemented in software (e.g. *A::g()* and *B::f()* in the "Instruction Memory" box in Fig.1) or in hardware (e.g. *A::f()* and *C::f()* FUs below the "Processor core" box).
- **Object instantiations:** These are specified in the *main()* function. A memory portion should be reserved for each instantiated object to store the values of its data items. This memory portion is allocated in a data memory (the gray box at the left-hand side of Fig.1, called OMU: Object Management Unit) that is accessible to the processor core as well as all FUs.
- **Method invocations:** The sequence of method invocations is specified in the *main()* function of the application. The executable code of this function comprises another part of the instruction memory (see Fig.1).

The processor core starts by reading the instructions specified in the *main()* function of the application. Whenever a method call instruction is read, the corresponding implementation is resolved and invoked. This may result in calling a software routine (e.g. *B::f()* in Fig.1) or activating an FU (e.g. *A::f()*). Each method implementation (be it in hardware or software) can also call other methods.

We view each method call as a network packet. Each method call is identified by a called method, a caller object and the parameters of call. Therefore, each packet comprises *mid*, *oid* and *params* to represent called method number, caller object number and call parameters.

The details on resolving method calls, passing parameters, synchronizing hardware and software, and other details of the architecture can be found in [10] and [11].

3.2 Data Prefetching Approach

In previous sections, we introduced the architecture of an OO-ASIP. The OMU in the mentioned architecture also contains a data cache and controls it to accelerate data access. According to the architecture of an OO-ASIP, the OMU is connected to the on-chip-network and can get all packets from the network and extract the called method and caller object IDs. Thus the OMU, as the cache controller, can be *aware* of the currently called method and also *aware* of the calling object and hence by our definition is *object aware*.

Class methods may be called on different objects, but the *unconditionally accessed* data fields of the called object are the same for all invocations. Here regarding the object awareness mentioned above, the OMU is aware of when a method is called by an object and can prefetch the unconditionally-accessed data fields of the called object to the cache.

This is a "directed" hardware prefetching policy compared to "Speculative" fetching of traditional caches. This avoids the instruction overhead of software prefetching mentioned before, while still offering its full potential. Also because we prefetch only the fields that will be certainly accessed, the penalty for useless prefetching can be eliminated. We will develop formulas for the hit-ratio of a cache with this prefetching mechanism in next section.

This prefetching scheme requires a static analysis to extract the unconditionally-accessed data fields of the called objects from the class method codes. Keeping the index of these object fields for each method has some area overhead. Although this overhead is dependant to the implementation method, it can be estimated using the number of all object field indices should be kept in the cache controller. In next sections we will present the number of data fields that should be kept for each benchmark we used.

This prefetching mechanism can be extended by prefetching the data items that are accessed with a probability higher than a threshold. Such information can be obtained from a dynamic analysis of the methods using some reasonable test vectors before synthesizing the OO-ASIP.

This allows a design trade-off: on one hand, *hit ratio* can be higher but, on the other hand, it consumes higher memory bandwidth (and more power) to prefetch the associated data, it requires a bigger cache controller to remember this per-method information, and moreover, some of these prefetched data may not be actually accessed resulting in some wasted resources.

In following sections we analyze this prefetching method and then present its experimental results, derived from the simulation of this mechanism on some object oriented benchmarks.

4 Mathematical Analysis

As mentioned before, the additional information of which we take advantage for data prefetching is the object data fields on which implemented methods work.

For a mathematical analysis we define A_M as the set of all data accesses of a certain method M during its execution. We divide this set into two disjoint subsets: $A_{M,un}$, and $A_{M,cnd}$, where $A_{M,un}$ is unconditionally data accesses and the latter is condition-

ally data accesses of method M (i.e. depends on some run-time condition such as data accesses in the body of an *if-then-else* statement). We have:

$$A_M = A_{M,un} \cup A_{M,cnd.} \quad (1)$$

The above factors are static in nature and can be determined from a static analysis of the class methods. We can use some other factors that have a dynamic nature and need a run-time analysis with some sample inputs to get computed. First, for a sample run of a certain method M, the probability that a certain data access operation "a" is performed shall be the ratio of the number of "a" accesses to the total number of data accesses performed by M:

$$P_{M,a,acc} = N_{M,a,acc} / N_{M,acc.} \quad (2)$$

Where $N_{M,a,acc}$ shows the number of times that "a" was accessed during the sample run of method *M*, and $N_{M,acc}$ shows the total number of data accesses that *M* performed in that same run. We also define $P_{M,a,avl}$ as the probability that a certain data *a* is available in the cache.

To express the hit ratio of a single class method *M,* in terms of the individual accesses, using the above definitions we have the following equation:

$$h_M = \sum P_{M,a,acc} \times P_{M,a,avl} . \quad (3)$$

The above equation can be broken into the sum of two summations over *unconditional* and *conditional* accesses as follows:

$$h_M = \sum_{a \in A_{M,un}} P_{M,a,acc} \times P_{M,a,avl} + \sum_{a \in A_{M,cond}} P_{M,a,acc} \times P_{M,a,avl}. \quad (4)$$

Note that the access pattern is independent of the cache operation, and therefore, $P_{M,a,acc}$ is the same for both traditional and *object-aware* caches. For an *object-aware* cache, however, the fetching policy ensures that all unconditional accesses hit the cache since their corresponding data is prefetched upon method invocation; hence, the $P_{M,a,avl}=1$ for all such accesses. This results in the following equation for the *object-aware* cache:

$$h_M = \sum_{a \in A_{M,un}} P_{M,a,acc} + \sum_{a \in A_{M,cond}} P_{M,a,acc} \times P_{M,a,avl.} \quad (5)$$

Assuming $P_{M,a,avl}$ is the same in the presence and absence of object-aware prefetching, subtracting equation 4 from 5 gives the improvement in the hit ratio caused by the *object-awareness* capability :

$$\Delta h_M = \sum_{a \in A_{M,un}} P_{M,a,acc} \times (1 - P_{M,a,avl}). \quad (6)$$

This shows that the *object-awareness* capability shall sure improve the hit ratio unless either M has no unconditionally-accessed data ($P_{M,a,acc}=0$), or the cache organization is such that unconditional accesses certainly hit the cache ($P_{M,a,avl}=1$).

Applications which run exclusively from the cache after an initial start-up transient would satisfy this criterion, but as mentioned before generally cache memory in embedded systems is limited and thus cache-awareness improves the hit ratio.

Up to this point we have talked about the hit-ratio during execution of a single method M. The program consists of a set of such method calls and hence the overall hit ratio can be computed as the following equation, where $P_{M,call}$ shows the probability of invoking method M:

$$h = \sum_{for\ all\ M} P_{M,call} \times h_{M.} \tag{7}$$

Finally, the hit-ratio improvement for the entire program execution shall be the following:

$$\Delta h_M = \sum_{for\ all\ M} P_{M,call} \left(\sum_{a \in A_{M,un}} P_{M,a,acc} \times (1 - P_{M,a,avl}) \right). \tag{8}$$

A detailed analysis of this approach can be found in [12].

5 Object Aware Cache Simulator and Experimental Environment

To verify the above theoretical analysis and evaluate the prefetching mechanism in practice, a custom cache simulator was developed and simulation experiments were completed on some publicly available benchmarks. For this simulation we have designed two tools: a simulator and a memory access trace generator.

The simulator is designed as a trace-driven simulator. A trace-driven cache simulator reads in a trace, that is, a record of the memory references extracted from a particular execution of a program. In this input file of the simulator, tags 0, 1 and 2 before a memory address entry indicate memory read, memory write and prefetch, respectively.

The designed simulator is parameterizable so that the same trace can be tried with different parameters for block size, associativity, cache size, replacement policy and prefetch strategy.

For extracting memory access trace of benchmarks we used PIN instrumentation tool [13]. Pin is a tool for the instrumentation of programs and provides a rich API that can be used to design tools for the instrumentation of programs. We used it to profile Linux executables for Intel IA-32 processors. Using Pin API we designed a tool that records all data memory accesses. A filter should be applied in the tool to prevent the recording of shared libraries data accesses. Memory accesses done by functions existing in included standard header files during program execution are ignored too, using filters we have developed.

To simulate prefetching, we have inserted an access to all unconditionally accessed class members at the beginning of all class methods manually, and separated them from function code by two flags. The tool saves all reference addresses, following a

proper prefix in an intermediate trace file. Then, another program gets the address traces from the trace generator and having the address of the flag, records memory addresses between two flags as prefetch address (with prefix 2) in the trace file.

According to the above descriptions, the *object awareness* is simulated by modifying the source of benchmarks (inserting an access to the object fields required by a method in the beginning of the method, separated by two flags), generating memory reference trace, processing the initial trace file, determining the prefetch addresses by detecting the flags and finally preparing the final trace file.

Table 1 shows the number of memory references for the benchmark programs. It also shows the number of data field indices (unconditionally-accessed object fields for all class methods) that we need to keep in the cache controller for each benchmark to perform prefetching.

Table 1. Number of memory references in the benchmarks

Program	# of memory references	# of class data field indices taht should be kept for prefetching
deltablue	79,367,672	181
ixx	24,050,809	701
eqn	33,507,014	398
richards	16,571,583	91
oocsim	15,429,414	57
v.44	10,381,496	121

All programs are written in C++. We suppose that these programs describe an OO-ASIP system and their class methods are software modules or hardware functional units. We have used four benchmarks from OOSCB C++ suit [14]; *richards*, *deltablue*, *eqn* and *ixx*. Because we should modify the source of benchmarks and insert prefetching flags in them, we didnot use very large benchmarks of the OOCSB suit, "*lcom*" and "*idl*". The following input parameters have been used for the benchmarks: 3000 for *deltablue*, 1 for *richards*, "*eqn.input.all*" for *eqn*, and a combination of *idl* files from [15] for *ixx*. The next programs are the source code of the cache simulator we used in this research (*oocsim*) and C++ code of a modem compression protocol (*v.44*). All benchmarks are compiled using *gcc 3.2* under linux/x86.

6 Simulation Results

The simulation results of the benchmarks in absence and presence of the proposed prefetching method are illustrated in Tables 2 and 3. The performance metric used for evaluating our prefetching method is the data cache miss ratio.

In all simulations, the cache structure is 4-way set-associative with 8 byte block size. *richards*, *v.44* and *oocsim* are small benchmarks and the range (and not the number) of their memory reference addresses is small. Thus for these three programs we have reduced the cache size to get valid results. Using smaller cache sizes for

richards benchmark (8 times smaller than the cache size used for *deltablue, eqn* and *ixx*) is also reported in some other papers, e.g. in [16]. Regarding the simulation results, presented in Tables 2 and 3, the proposed prefetching method enhances the data cache behavior and on average causes the miss ratio to reduce 3.7 times (or by 73%).

Table 2. Simulation results for the large benchmarks. Miss ratio(%) with *Object-Aware* prefetching and without prefetching.

Program	8 K		4 K		1 K	
	No Prefetching	OA Prefetching	No Prefetching	OA Prefetching	No Prefetching	OA Prefetching
deltablue	4.6	0.9	6.1	1.03	7.99	2.75
ixx	1.9	0.82	4.89	2.05	8.88	5.19
eqn	1.07	0.67	4.27	3.65	11.66	10.17

Table 3. Simulation results for the small benchmarks. Miss ratio(%) with *Object-Aware* prefetching and without prefetching.

Program	1 K		1/2 K		1/8 K	
	No Prefetching	OA Prefetching	No Prefetching	OA Prefetching	No Prefetching	OA Prefetching
richards	0.05	0.02	1.99	0.71	13.03	8.29
oocsim	1.61	0.09	3.18	0.85	7.51	5.07
v.44	9.83	4.59	21.79	15.7	23.81	16.04

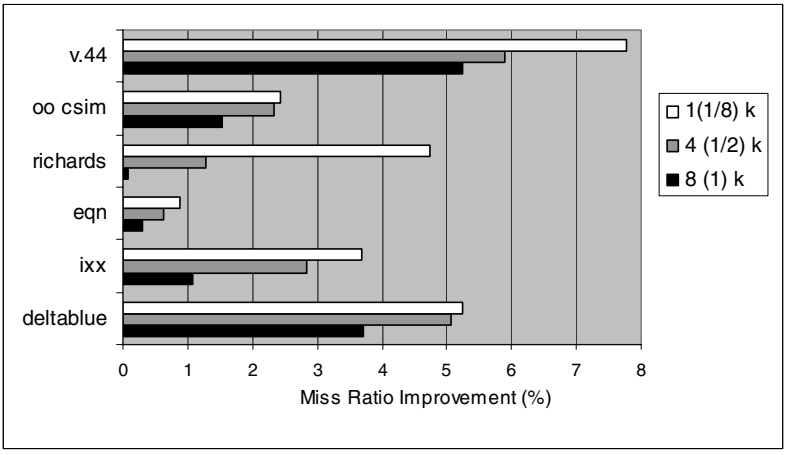


Fig. 2. Miss ratio improvement vs. cache size

Fig.2, presents the miss ratio improvement that is the difference between miss ratios (or hit ratios) with and without prefetching, for various cache sizes, in the benchmarks. In this figure, the cache size for small benchmarks (*v.44, oocsim* and *richards*) has been set 8 times smaller than the size used for the large benchmarks. Although the amount of hit ratio improvement is application dependent, if we reduce the size of the cache, the effect of the proposed prefetching technique will increase.

According to the results, we can get a higher hit ratio in a cache with defined size or can get a defined hit ratio in a smaller cache. This effect is useful because of limited cache size in the embedded systems.

In Fig.3 and 4 and Tables 4 and 5, the results of comparison between *one block look-ahead(OBL)* and *object-aware* prefetching techniques for *deltablue* and *richards* benchmarks are illustrated. We selected OBL because it is The tables and figures present the miss ratio (%) of each prefetching technique. Regarding the results, having some hardware overhead for keeping per-method information, object-aware prefetching shows better results than *one block look-ahead* prefetching technique.

While we prefetch only unconditionally accessed object fields needed by a class method (hardware functional unit or software module) there is no power dissipation because of wrong prefetchings. But in the implementation of this method, a trade-off may be needed between hit ratio improvement and area which we need for keeping per-method information.

The number of data field indices we need to keep in the cache controller for each benchmark, which is shown in Table 1, can be used to estimate the area overhead of this method. According to Table 1, this overhead will be reasonable for most of benchmarks. For some benchmarks like *ixx,* which have higher overhead than the others, we can perform a run-time analysis using some test vectors and keep the information for most called methods. For example, only 41% of the *ixx* class methods were called in the simulation reported in Table 2. Eliminating the rarely-used methods will decrease the area overhead (about 50% for *ixx)* while having no effect on the performance.

We are currently preparing an implementation of our prefetching scheme, so that the design trade-offs (hit ratio improvement and power/area) can be performed in practice.

Table 4. One Block Look-Ahead vs. Object-Aware Prefetching for *deltablue* (Miss Ratio)

Prefetching Mechanism	1k	2k	4k	8k
No Prefetching	7.99	6.6	6.1	4.6
OBL	5.16	5.11	4.57	4.1
Object Aware	2.75	1.6	1.03	0.9

Table 5. One Block Look-Ahead vs. Object-Aware Prefetching for *richards* (Miss Ratio)

Prefetching Mechanism	1/8 k	1/4 k	1/2 k	1k
No Prefetching	13.03	6.69	1.99	0.05
OBL	12.16	6.42	1.84	0.04
Object Aware	8.29	5.4	0.71	0.02

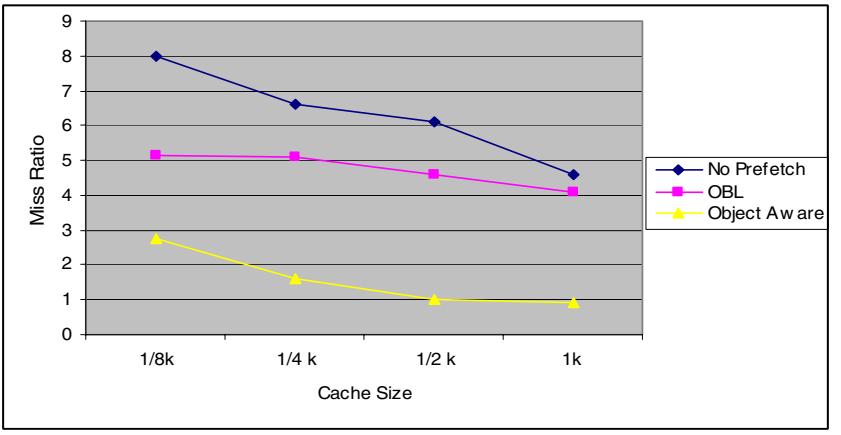


Fig. 3. One block look-ahead vs. object-aware prefetching schemes for *deltablue* (Miss Ratio)

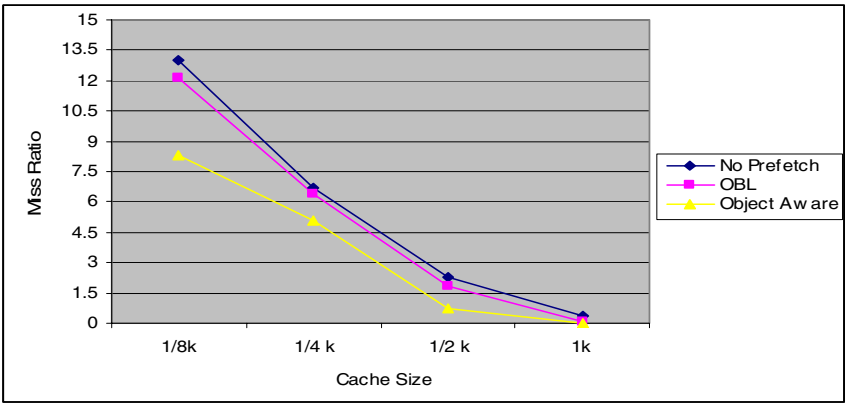


Fig. 4. One block look-ahead vs. object-aware prefetching schemes for *richards* (Miss Ratio)

7 Summary and Conclusions

In this paper, we presented a mechanism and the simulation results for an application-specific hardware-driven prefetching scheme to improve data cache performance. Some previous works have used hardware data prefetching, software data prefetching or a combination of them, but in our technique, taking advantage of architectural properties of OO-ASIPs, the cache controller knows the currently running method and the called object and can therefore prefetch the unconditionally-accessed data fields of the called object to the cache.

The simulation is done using PIN instrumentation tool and on some object-oriented benchmarks. Simulation results confirm analytical results and prove that this technique enhances the data cache behavior. The proposed prefetching approach reduces the miss

ratio by 73% in average in the used benchmarks. The amount of hit ratio enhancement is application dependent but the hit ratio increment is higher in small cache sizes.

Also the simulation results show that the proposed method has higher hit ratios than *one block look-ahead* prefetching technique. One important point in embedded system caches is their limited size. Due to this limited size, this proposed prefetching mechanism will be useful to increase the hit ratio of a defined cache or achieve the same hit ratio in a cache having smaller size.

While we prefetch only unconditionally accessed object fields needed by a hardware functional unit or software module, there is no power dissipation caused by wrong prefetchings. The area overhead for keeping prefetching information can be estimated by the number of unconditionally-accessed data fields in all class methods in the programs. The number of these fields is calculated for the benchmarks and is reasonable for most of them. If the overhead is high for a benchmark, we can reduce it by eliminating the information for rarely-used and unused methods using a run-time analysis.

References

[1] Hennessy, J., and D. Patterson. Computer Architecture: a Quantitative Approach (3rd Edition). Morgan Kaufmann, 2003.

[2] K.K. Chan, et al., "Design of the HP PA 7200 CPU," *Hewlett-Packard Journal,* vol. 47, no. 1, February 1996.

[3] 42. Przybylski, S., "The Performance Impact of Block Sizes and Fetch Strategies," *Proc. 17th ISCA*, Seattle, WA, May 1990, pp. 160-169.

[4] Dahlgren, F., M. Dubois and P. Stenstrom, "Fixed and Adaptive Sequential Prefetching in Shared-memory Multiprocessors," *Proc. International Conference on Parallel Processing*, St. Charles, IL, August 1993, pp. I-56-63.

[5] Chen, T-F and J-L. Baer, "Effective Hardware-Based Data Prefetching for High PerformanceProcessors," *IEEE Transactions on Computers*, Vol. 44, No. 5, May 1995, pp. 609-623.

[6] Joseph, D. and D. Grunwald. "Prefetching using Markov Predictors," *Proc. 24th ISCA*, Denver, CO, June 1997, pp. 252-263.

[7] S.P. VanderWiel, and D.J. Lilja, "Data prefetch mechanisms," *ACM Computing Surveys,* vol. 32, issue 2, pp. 174-199, June 2000.

[8] Chen, T-F and J. L. Baer, "A Performance Study of Software and Hardware Data Prefetching Schemes," *Proc. of the 21st Annual International Symposium on Computer Architecture, Chicago*, IL, April 1994, pp. 223-232.

[9] http://research.sun.com/projects/dashboard.php?id=13 , Mayhem research group, March 2005.

[10] M. Goudarzi, S. Hessabi, and A. Mycroft, "Object-oriented ASIP design and synthesis," *Proc. of Forum on specification and Design Languages (FDL'03),* September 2003.

[11] M. Goudarzi, S. Hessabi, and A. Mycroft, "No-overhead polymorphism in network-on-chip implementation of object-oriented models," *Proc. of Design Automation and Test in Europe (DATE'04),* February 2004.

[12] M. Goudarzi, S. Hessabi, A. Mycroft "Object-aware Cache: Higher Cache Hit-ratio in Object- Oriented ASIPs", Proc. of *IEEE Canadian Conference on Electrical and Computer Engineering (CCECE' 04)*, Ontario, Canada, May 2004.

[13] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, Kim Hazelwood. "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation," *Programming Language Design and Implementation (PLDI),* Chicago, IL, June 2005.
[14] OOCSB C++ Benchmark suit, http://www.cs.ucsb.edu/labs/oocsb.
[15] http://www.gnuenterprise.org/~neilt/code/html/files.html, March 2005.
[16] Yul Chu, Ito, M.R., "An efficient instruction cache scheme for object-oriented languages," *Performance, Computing and Communications, IEEE International Conference on*, pp. 329 -336, April 2001.
[17] Dinero cache simulator, http://www.cs.wisc.edu/~markhill/DineroIV.
[18] D. Bernstein, C. Doron, and A. Freund, "Compiler techniques for data prefetching on the PowerPC," *Proc. of International Conf. on Parallel Architectures and Compilation Techniques,* June 1995.

Targeted Data Prefetching

Weng-Fai Wong

Department of Computer Science, and Singapore-MIT Alliance,
National University of Singapore,
3 Science Drive 2, Singapore 117543
wongwf@comp.nus.edu.sg

Abstract. Given the increasing gap between processors and memory, prefetching data into cache becomes an important strategy for preventing the processor from being starved of data. The success of any data prefetching scheme depends on three factors: *timeliness*, *accuracy* and *overhead*. In most hardware prefetching mechanism, the focus has been on accuracy - ensuring that the predicted address do turn out to be demanded in a later part of the code. In this paper, we introduce a simple hardware prefetching mechanism that targets *delinquent loads*, i.e. loads that account for a large proportion of the load misses in an application. Our results show that our prefetch strategy can reduce up to 45% of stall cycles of benchmarks running on a simulated out-of-order superscalar processor with an overhead of 0.0005 prefetch per CPU cycle.

1 Introduction

The growing gap between processors and memory has led to performance not growing with improvements in processor technology in a commensurate way. The introduction of caching has alleviated somewhat the problem though not eliminating it entirely. In many state-of-the-art processors, a miss that requires going to the memory can cost hundreds of cycles. This problem can only be made worse by the introduction of chip multiprocessors which only serve to increase the demand for the timely delivery of data.

Data prefetching is an important strategy for scaling this so-called memory wall. There are two main classes of data prefetching strategies. In the pure hardware approach, additional hardware is added to the processor that monitors the execution of the program, launching prefetch requests when the opportunity arises. The key advantage of this approach is its transparency. Neither the source code nor any special knowledge about the application is needed. In the hybrid software-hardware approach, the hardware exposes its prefetching mechanism to the compiler or programmer. The compiler is then suppose to take advantage of some knowledge about the application to maximize the utility of these hardware mechanisms. An example of such mechanisms could be a simple prefetch instruction which serves as a hint to the processor to promote a piece of datum higher up the memory hierarchy. The disadvantage of the hybrid approach is that the source code of the application is often needed for the compilation process. In this paper, we are concerned with hardware prefetching.

T. Srikanthan et al. (Eds.): ACSAC 2005, LNCS 3740, pp. 775–786, 2005.

For any prefetch mechanism to work, three criteria have to be met. The prefetch has to be *timely*. Data have to be brought in just in time to met the demand of it. Bringing in data too early risks having it being evicted from the cache when it is needed, and of course, performing the prefetch after the original load has been issued renders the prefetch useless. Any prefetch requires the ability to predict the address of the data that will be needed ahead of its time. This prediction has to be *accurate* in order that the prefetch not be wasted. Finally, the *overhead* associated with performing an overhead has to be small. This overhead may be measured in terms of the amount of hardware needed to support a hardware prefetcher or the pressure exerted by the prefetches on the processor's resource.

Most prefetch schemes suggested in the literature thus far has been focused on the issue of accuracy. In this paper, we propose a strategy for data prefetching that addresses the question of overhead without sacrificing accuracy and timeliness. The key insight exploited by this mechanism is that most of the cache misses experienced by an application are due to a small number of *delinquent loads* [9,27]. By focusing the prefetch effort on these delinquent loads, one can reduce the number of prefetches issued without sacrificing much in terms of accuracy or timeliness.

The rest of the paper is organized as follows. Section 2 will survey the related works in the field. Section 3 will introduce our proposed mechanism. This is followed by our experimental setup, the results and a discussion of the results. The paper ends with a conclusion and some ideas for future works.

2 Related Work

There has been a lot of research into data prefetching [30]. Early work on software prefetching focused on prefetching for data intensive loops [4,19]. With the proper analysis, regularly nested loops can benefit tremendously from data prefetching. [26]. Beyond loops, prefetching for procedure parameters [21] and recursive data structures [23] have also been proposed. Karlsson, Dahlgren, and Stenstrom [17] proposed the use of prefetch arrays while VanderWiel and Lilja proposed a *data prefetch controller* (DPC) [29]. More recently, there has been some interest in using a 'helper' hardware thread to prefetch ahead of the main thread [18]. Rabbah et. al. [28] proposed the use of spare resources in an EPIC processor to accomodate the prefetch thread instead of using a hardware thread.

Pure hardware prefetching schemes includes Jouppi's "stream buffers" [16], Fu and Patel's prefetching for superscalar and vector processors [11,12], and Chen and Baer's lookahead mechanism [7] and known as the *Reference Prediction Table* (RPT) [8]. Mehrota [25] proposed a hardware data prefetching scheme that attempts to recognize and use recurrent relations that exist in address computation of link list traversals. Extending the idea of correlation prefetchers [6], Joseph and Grunwald [15] implemented a simple Markov model to dynamically prefetch address references. More recently, Lai, Fide, and Falsafi [20] proposed a hardware mechanism to predict the last use of cache blocks.

On load address prediction, Eickemeyer and Vassiliadis first proposed the use of a stride predictor to predict the address of a load [10]. However, it is not a prefetch as the prediction is verified before any memory system activity is triggered. This idea was extended by others by means of operand-based predictors [2] and last-address predictors [22]. The use of a stride predictor in the launch of actual prefetches was proposed by Gonzalez and Gonzalez [14]. A more elaborate scheme to enhance prediction accuracy was proposed by Bekerman et. al. [3]. The latter also expressed concern on the negative effects spurious prefetches have on the cache and made special provisions to avoid these from polluting the cache.

The delinquent load identification hardware scheme is similar to the local hit-miss predictor proposed by Yoaz et. al. [31]. However, they used it to predict the dynamic latencies of loads. The predicted load latencies are used to help the instruction scheduler more accurately schedule load instructions. For their purpose, a higher degree of accuracy is required as compared to ours which is used to spot potential delinquent loads. Furthermore, we applied the filter only to misses as we only launch prefetches on level one data cache misses. As far as we know, the use of the simple delinquent load identification scheme for the purpose of prefetching is novel.

3 Proposed Prefetching Mechanism

Data prefetching involves two important decisions, namely:

- *When* should a prefetch be triggered? It is not desirable to launch prefetches too early or too late. Neither is it desirable to launch more prefetches than what is required because there are fixed overheads associated with the processing of each prefetch.
- *What* should be prefetch? It is necessary to predict the address(es) of the data items required by the processor in the near future. As with many successful applications of predictions, here, past history serves as a good indicator for future use. The accuracy of the prediction is key to the success of such schemes. Some prefetch schemes performs pre-computation instead of prediction [1]. These schemes are generally much more accurate then mere prediction. However, the hardware investments for this can be substantial as portions of the pre-computation may turn out to be redundant.

We propose a prefetch scheme that addresses these issues as follows. The first key insight our scheme takes advantage of is that only a small number of load instructions, known as delinquent loads, are responsible for a large number of the cache misses. It is for these loads that we will launch prefetches. In order to identify delinquent loads, we hash the program counter for a load instruction into a table of saturating counters. Let PC stand for the program counter. Let h(PC) be the 3-bit saturating counter obtained by hashing PC into the delinquent load table, a table of saturating counters. If the counter is non-zero, then the load is identified as a delinquent load. The counter is updated as follows: if the current

load misses the cache, the counter is incremented (up to 7, beyond which it cannot be incremented further). If the current load hits the cache, the counter is decremented (down to zero, below which it cannot be decremented further). The main idea is for the counter to track if, in the recent history of this particular load, there were more misses than hits. If so, it is classified as delinquent. The length of the counter affects how much of the past history is considered. The choice of 3-bit counters is along the lines of the experiences gained in the branch prediction community.

The proposed architecture is shown in Fig. 1. In an out-of-order processor, a *load-store queue* serves as the reorder buffer for loads and stores. In our proposed architecture, we require the program counter to be noted alongside the load requests. When a load reaches the head of the load-store queue for processing, in parallel with cache lookup, a delinquent table lookup is performed. If it is known that the load misses the (L1) cache, a prediction is performed. Not shown in Fig. 1 is how the delinquent load table is also updated according to the description above. The predicted load address forms a prefetch request that enters the prefetch queue. This queue competes with the main load-store queue for service by the memory ports. Prefetch requests that hit the cache, however, are discarded.

When a delinquent load misses, prefetching may then occur. In order to contain the amount of resources needed to realize the prefetch mechanism, we chose a prediction based scheme. In particular, we chose to use a hybrid *stride*

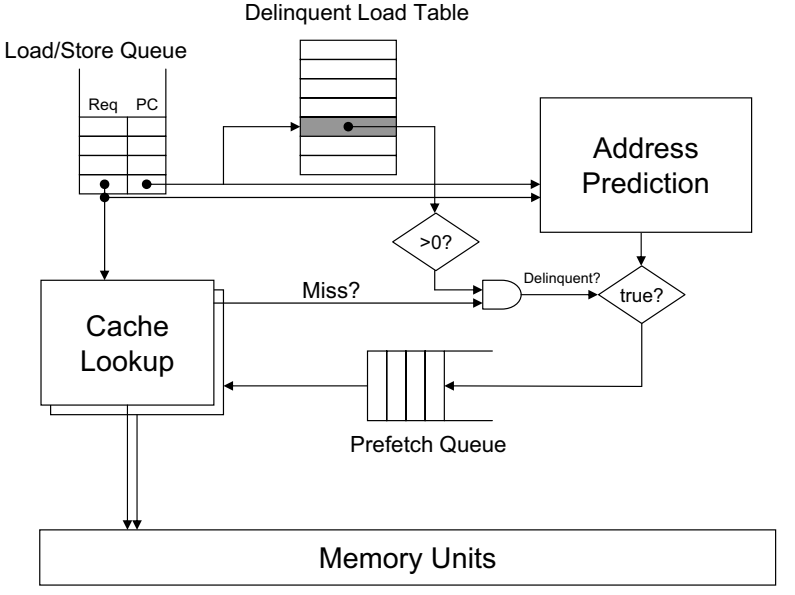


Fig. 1. Proposed prefetching architecture

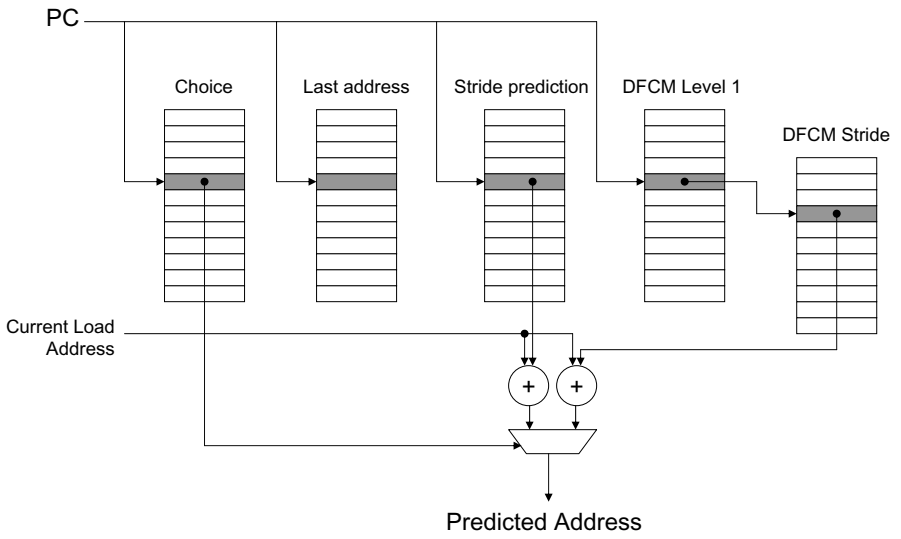


Fig. 2. Hybrid load address predictor used

and *differential finite context method* predictor [13]. This predictor is shown in Fig. 2. First, a last value table records the previous address which the current load accessed. After the current prediction is completed, it is updated with the current load address. The stride predictor records most recently seen stride for the current load. Its predicted address is formed by adding the last seen stride to the current load address. The predictor is updated by recording the difference between the current address and the last seen address of this load which is recorded in the last address table. The differential finite context method is another stride predictor that makes use of a two-level table. In doing so, it is able to account for the context information under which a stride was last seen. We refer the reader to the original paper on DFCM [13] for a detailed discussion of the rationale and working of this state-of-the-art predictor. The last address table is also needed for the update of this predictor.

The idea of hybridizing two predictors comes from the branch prediction community [24]. Based on the past history of success, a table of saturating counters is checked to see which of the two predictors were more successful for a particular load and to use that predictor's prediction in the current prefetch. To update the predictor choice table, the last seen address is added to the strides predicted by both predictors. These are the previous predictions made by the predictors. These are then compared with the current load address. If the stride predictor is correct, the counter is decremented. If the DFCM predictor is correct, the counter is incremented. The next choice of predictor will depend on whether the counter is greater or less than zero. If it is exactly at zero, an arbitrary choice is made. In our experiments, the prediction of the DFCM is taken.

4 Experiment Setup

We realized the proposed architecture through modifying the SimpleScalar simulator for an out-of-order superscalar processor [32]. The machine parameters used in our experiments are listed in Table 1. We used a delinquent load table of 2,048 3-bit counters. All other tables, i.e. the last value table (which is 4 bytes per entry), the stride predictor (2 bytes per entry), DFCM level 1 table (10 bits per entry), DFCM level 2 stride table (2 bytes per entry), and the choice predictor (3 bits per entry), are 1,024 entries each. The total table size is about 10.3 Kbytes. Note that this is significantly less than the 16 Kbyte L1 data cache especially when tag storage is also considered.

We evaluated the performance of our proposed prefetching scheme using benchmarks from the SPEC [33] and the Olden [5] benchmark suite. In detail, the benchmarks used were a Lisp interpreter (`130.li`), a JPEG encoder (`132.ijpeg`), gzip compression (`164.gzip`), quantum chromodynamics (`168.wupwise`), shallow water modeling (`171.swim`), a multigrid solver (`172.mgrid`), a neural network (`179.art`), combinatorial optimization (`181.mcf`), seismic wave propagation simulation (`183.equake`), computational chemistry (`188.ammp`), word processing (`197.parser`), an object oriented database (`255.vortex`), BZIP2 compression (`256.bzip2`), electromagnetic wave propagation in a 3D object (`em3d`), and the traveling salesman problem (`tsp`).

Table 1. Simulated out-of-order machine used in the experiments

Parameter	Value
Instruction fetch queue size	4 instructions
Branch predictor	Combined predictor
Decode width	4 instructions
Issue width	4 instructions
Commit width	4 instructions
Load-store queue length	8 requests
Prefetch queue length	8 requests
Memory ports	2 or 4
Integer ALU	4
Integer multiplier	1
FP ALU	4
FP multiplier	1
Number of registers	32
L1 inst cache	16K, 32-byte, direct, LRU
L1 data cache	16K, 32-byte, 4-way, LRU
L1 hit latency	1 cycle
L2 data cache	256K, 64-byte, 4-way, LRU
L2 hit latency	10 cycle
L2 miss latency	min. 100 cycle

As a comparison, we implemented the standard RPT scheme that attempts prefetching on every load instruction. In addition, we also implemented Joseph and Grunwald's Markovian prefetcher [15]. The latter will launch four prefetches for each delinquent load. These four addresses are basically the last four addresses previously accessed by a particular load and missed the cache.

5 Performance Evaluation Results

Fig. 3 shows the result of our experiments for a machine with two memory ports. It shows the ratio of simulated machine cycles of the RPT, Markovian (denoted by 'JosGrun') and our proposed prefetch schemes against a baseline machine that do not perform any prefetching. In the best case (`179.art`), it took only 56% of the total machine cycles of the baseline to complete the application with our prefetch scheme. In the worst case, however, it took 13% more cycles than the baseline to complete with our prefetching scheme. This happens when the prediction turned out to perform badly. Overall, our proposed scheme reduced the baseline cycle count by about 12.5%. This is marginally better than RPT. On the other hand, except in a few instances, the Markovian predictor performed poorly.

For a machine with four memory ports, the results are shown in Fig. 4. They are similar to that for that for two memory ports. The noticable difference is that here RPT performs better for our scheme.

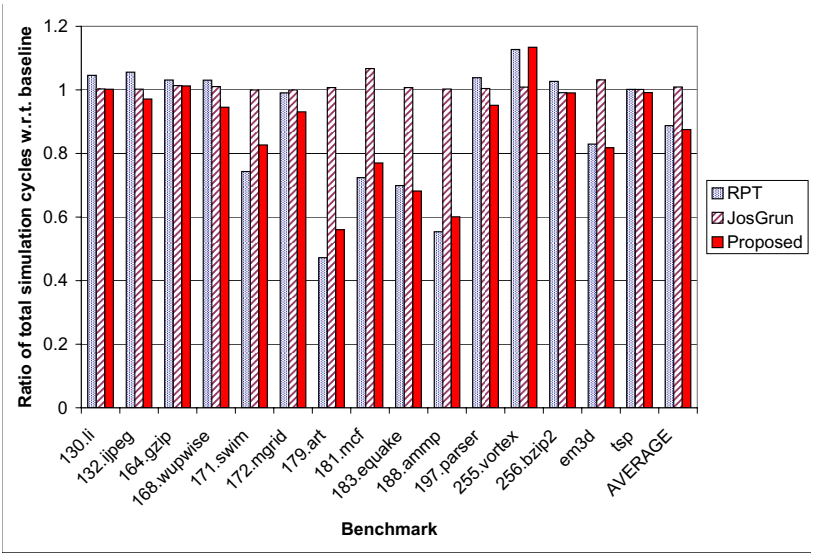


Fig. 3. Performance of various prefetching schemes on a simulated out-of-order superscalar processor with two memory ports

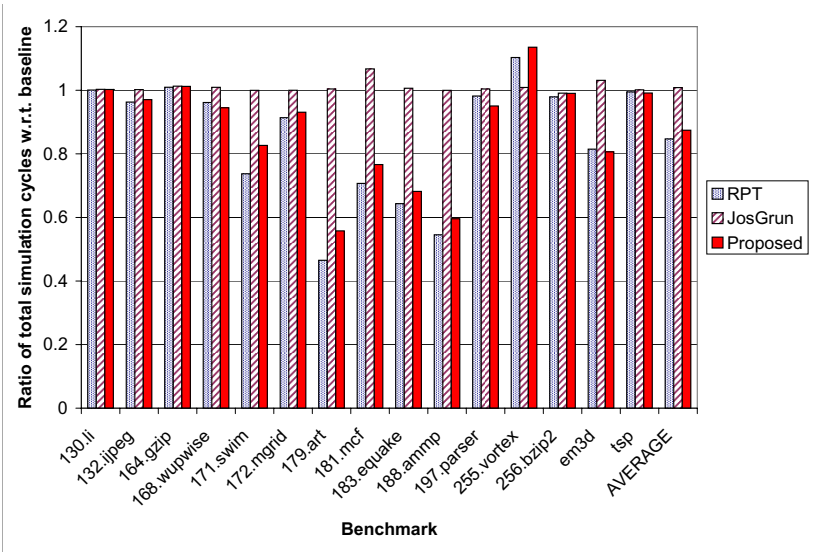


Fig. 4. Performance of various prefetching schemes on a simulated out-of-order superscalar processor with four memory ports

RPT performs better because it covers all loads, which includes all delinquent loads, whereas our delinquent load predictor can still at times miss out on certain delinquent load. This improved performance for RPT, however, comes at a price. Table 2 shows the number of prefetches launched by RPT, the Markovian predictor and our scheme. The second column is the ratio of the number of prefetches launched and the total machine cycles for the application running on the baseline machine with four memory ports. This highlights the key point of the paper: with our delinquent load identification scheme, we can achieve a performance gain competitively comparable with RPT but with *only* 7% of the prefetches launched by RPT. This is significant in processor designs where there are more constraints on resources or where power-energy is an issue.

Completing the execution of an application early is often advantageous from a total energy consumption perspective. However, each prefetch consumes energy, thus our proposed prefetching scheme will save a significant amount of energy most of the time. Furthermore, in many high performance processors where heating is a serious concern, the lower amount of additional processor activity due to prefetching is also conducive to maintaining the temperature profile of the processor.

Table 3 shows the number of load instructions, level 1 data cache miss rate and the percentage loads identified as delinquent using our scheme. The delinquency rate is generally significantly lower than the miss rate. In some instances, the miss rate is lower. However, the miss rate is computed over all memory references including writes so the two values cannot be directly compared but merely serves as an indication of the selectivity of our scheme.

Table 2. Number of prefetches launched (in millions). The ratio of the number of prefetches launched by a particular scheme over that for RPT is shown inside parenthesis.

Benchmark	RPT	Fract. of Total Cyc.	JosGrun	Proposed Scheme
130.li	177	41.88%	17 (9.60%)	5 (2.82%)
132.ijpeg	12,405	44.88%	346 (2.79%)	103 (0.83%)
164.gzip	528	29.52%	98 (18.56%)	45 (8.52%)
168.wupwise	5,255	32.96%	281 (5.35%)	47 (0.89%)
171.swim	12,751	18.91%	1,939 (15.21%)	526 (4.13%)
172.mgrid	420	51.13%	10 (2.38%)	2 (0.48%)
179.art	4,998	10.88%	2,322 (46.46%)	495 (9.90%)
181.mcf	427	17.37%	318 (74.47%)	88 (20.61%)
183.equake	34,739	34.60%	6,370 (18.34%)	1,638 (4.72%)
188.ammp	905	9.81%	865 (95.58%)	185 (20.44%)
197.parser	1,579	31.29%	254 (16.09%)	108 (6.84%)
255.vortex	6,936	21.34%	562 (8.10%)	365 (5.26%)
256.bzip2	1,548	40.15%	57 (3.68%)	26 (1.68%)
em3d	96	11.04%	59 (61.46%)	16 (16.67%)
tsp	8,723	3.87%	404 (4.63%)	110 (1.26%)
Average			(25.51%)	(7.00%)

Table 3. Delinquency in our benchmarks

Benchmark	Total num. miss rate	Baseline D1 of loads	Delinquency
130.li	211	0.93%	1.30%
132.ijpeg	14,594	0.55%	0.39%
164.gzip	595	2.70%	3.63%
168.wupwise	6,600	0.40%	0.41%
171.swim	13,340	4.26%	1.98%
172.mgrid	426	0.70%	0.26%
179.art	6,134	9.18%	4.03%
181.mcf	594	8.10%	8.01%
183.equake	36,772	3.81%	2.89%
188.ammp	1,015	12.33%	9.29%
197.parser	2,151	1.86%	2.42%
255.vortex	8,181	0.84%	2.82%
256.bzip2	1,999	0.79%	0.65%
em3d	132	7.01%	0.63%
tsp	10,785	0.44%	0.48%

6 Conclusion

In this paper, we introduced an architecture for hardware prefetching that targets delinquent loads. When coupled with a hybrid load address predictor, our experiments showed that the prefetching scheme can reduce the total machine cycles by as much as 45% by introducing a low overhead. For the latter, the ratio of prefetches launched by our proposed scheme over the total machine cycles for the baseline out-of-order machine with four memory port ranges from 0.05% to 3.6%. This contrasts with the 3.8% to 51% overhead for RPT which achieves similar performance gains. We therefore argue that our proposed scheme fulfils the three criteria of a good prefetch scheme discussed in the Introduction, namely timeliness, accuracy and low overhead.

The key contribution of this work is in pointing out that even with a simple scheme, prefetching can be targeted very specifically to the load instructions that matter, and this yields significant practical benefits.

On a final note, the proposed architecture is general enough to work with any load address prediction scheme. It would be interesting to see if other prediction schemes, perhaps even ones that are uniquely designed for different applications so as to optimize area-performance, say, can benefit from it.

References

1. Annavaram, M., Patel, J.M., Davidson, E.S.: Data prefetching by dependence graph precomputation. In: Proceedings of the 28th Annual International Symposium on Computer Architecture. (2001) 52–61
2. Austin, T.M., Sohi, G.S.: Zero-cycle loads: Microarchitecture support for reducing load latency. In: Proceedings of the 28th International Symposium on Microarchitecture. (1995) 82 – 92
3. Bekerman, M., Jourdan, S., Ronen, R., Kirshenboim, G., Rappoport, L., Yoaz, A., Weiser, U.: Correlated load-address predictors. In: Proceedings of the 26th Annual International Symposium on Computer Architecture. (1999) 54–63
4. Callahan, D., Kennedy, K., Porterfield, A.: Software prefetching. In: Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems. (1991) 40–52
5. Carlisle, M.C.: Olden: Parallelizing Programs with Dynamic Data Structures on Distributed-Memory Machines. PhD, Princeton University Department of Computer Science (1996)
6. Charney, M., Reeves, A.: Generalized correlation based hardware prefetching. Technical Report EE-CEG-95-1, Cornell University (1995)
7. Chen, T.F., Baer, J.L.: A performance study of software and hardware data prefetching schemes. In: Proceedings of International Symposium on Computer Architecture. (1994) 223 – 232
8. Chen, T.F., Baer, J.L.: Effective hardware-based data prefetching for high-performance processor computers. IEEE Transactions on Computers **44-5** (1995) 609 – 623
9. Collins, J.D., Wang, H., Tullsen, D.M., Hughes, C., Lee, Y.F., Lavery, D., Shen, J.P.: Speculative precomputation: Long-range prefetching of delinquent loads. In: Proceedings of the 28th International Symposium on Computer Architecture. (2001) 14–25

10. Eickemeyer, R.J., Vassiliadis, S.: A load-instruction unit for pipelined processors. IBM Journal of Research and Development **37** (1993) 547–564
11. Fu, J.W.C., Patel, J.H.: Data prefetching strategies for vector cache memories. In: Proceedings of the International Parallel Processing Symposium. (1991)
12. Fu, J.W.C., Patel, J.H.: Stride directed prefetching in scalar processors. In: Proceedings of the 25th International Symposium on Microarchitecture. (1992) 102 – 110
13. Goeman, B., Vandierendonck, H., Bosschere, K.D.: Differential FCM: Increasing value prediction accuracy by improving table usage efficiency. In: Proceedings of the 7th International Sysmposium on High-Performance Computer Architecture. (2001) 207 – 216
14. Gonzalez, J., Gonzalez, A.: Speculative execution via address prediction and data prefetching. In: Proceedings of the 11th International Conference on Supercomputing. (1997) 196–203
15. Joseph, D., Grunwald, D.: Prefetching using markov predictors. In: Proceedings of the 24th International Symposium on Computer Architecture. (1997) 252 – 263
16. Jouppi, N.P.: Improving direct-mapped cache performance by the addition of a smally, fully associative cache and prefetch buffers. In: Proceedings of the 17th International Symposium on Computer Architecture. (1990) 364 – 373
17. Karlsson, M., Dahlgren, F., Stenstrom, P.: A prefetching technique for irregular accesses to linked data structures. In: Proceedings of the 6th International Sysmposium on High-Performance Computer Architecture. (2000) 206 – 217
18. Kim, D., Liao, S.S., Wang, P.H., del Cuvillo, J., Tian, X., Zou, X., Wang, H., Yeung, D., Girkar, M., Shen, J.P.: Physical experimentation with prefetching helper threads on Intel's hyper-threaded processors. In: Proceedings of the International Symposium on Code Generation and Optimization. (2004) 27–38
19. Klaiber, A.C., Levy, H.M.: An architecture for software-controlled data prefetching. In: Proceedings of the 18th International Symposium on Computer Architecture. (1991) 43 – 53
20. Lai, A.C., Fide, C., Falsafi, B.: Dead-block prediction and dead-block correlation prefetchers. In: Proceedings of the International Parallel Processing Symposium. (2001) 144 – 154
21. Lipasti, M., Schmidt, W., Kunkel, S., Roediger, R.: Spaid: Software prefetching in pointer and call-intensive environment. In: Proceedings of the 28th International Symposium on Microarchitecture. (1995) 231 – 236
22. Lipasti, M.H., Wilkerson, C.B., Shen, J.P.: Value locality and load value prediction. In: Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems. (1996) 138–147
23. Luk, C.K., Mowry, T.: Compiler-based prefetching for recursive data structures. In: Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems. (1996) 222 – 233
24. McFarling, S.: Combining branch predictors. Technical Report TN-36, DEC WRL (1993)
25. Mehrota, S., Luddy, H.: Examination of a memory classification scheme for pointer intensive and numeric programs. Technical Report CRSD Tech. Report 1351, CRSD, University of Illinois (1995)
26. Mowry, T.C., Lam, M.S., Gupta, A.: Design and evaluation of a compiler algorithm for prefetching. In: Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems. (1992) 62 – 73

27. Panait, V.M., Sasturkar, A., Wong, W.F.: Static identification of delinquent loads. In: Proceedings of the International Symposium on Code Generation and Optimization. (2004) 303–314
28. Rabbah, R.M., Sandanagobalane, H., Ekpanyapong, M., Wong, W.F.: Compiler orchestrated prefetching via speculation and predication. In: Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems. (2004) 189–198
29. Vanderwiel, S., Lilja, D.: A compiler-assisted data prefetch controller. In: Proceedings of the International Conference on Computer Design. (1999) 372 – 377
30. Vanderwiel, S., Lilja, D.J.: Data prefetch mechanisms. ACM Computing Survey **32** (2000) 174 – 199
31. Yoaz, A., Erez, M., Ronen, R., Jourdan, S.: Speculation techniques for improving load related instruction scheduling. In: Proceedings of the 26th Annual International Symposium on Computer Architecture. (1999) 42–53
32. The SimpleScalar toolkit.: http://www.simplescalar.com. (2005)
33. The SPEC benchmarks.: http://www.spec.org. (2000)

Area-Time Efficient Systolic Architecture for the DCT

Pramod Kumar Meher

School of Computer Engineering,
Nanyang Technological University, Singapore 639798
aspkmeher@ntu.edu.sg,
http://www.ntu.edu.sg/home/aspkmeher/

Abstract. A reduced-complexity algorithm and its systolic architecture are presented for computation of the discrete cosine transform. The proposed scheme not only leads to a fully-pipelined regular and modular hardware, but also offers significantly higher throughput, lower latency and lower area-time complexity over the existing structures. The proposed design is devoid of complicated input/output mapping and complex control structure. Moreover, it does not have any restriction on the transform-length and it is easily scalable for higher transform-length as well.

1 Introduction

The discrete cosine transform (DCT) plays a key function in many signal and image processing applications, especially for its near optimal beahviour in transform coding [1]. Fast computation of the DCT is considered as a highly desirable and challenging task because it is computationally intensive and frequently encountered in various real-time applications. Many algorithms and architectures are, therefore, suggested for computing the DCT in dedicated VLSI [2] to meet the performance requirement of time-constrained applications and area-constraint embedding environments. Amongst the existing VLSI systems, systolic architectures have been extensively popular owing not only to the simplicity of their design using repetitive identical processing elements (PE) having regular and local interconnections and rhythmic processing; but also for the potential of using high level of pipelining in small chip-area with low power consumption inherent with the structure [3]. Several algorithms and systolic architectures are, therefore, reported in the literature for efficient VLSI implementation of the DCT [4-9], where the computing structures mainly differ in terms of their hardware-complexity, time-complexity, control structure, and I/O requirement. This paper aims at a novel systolic array for the DCT which would provide higher-throughput, lower-latency and lower area-time complexity over the existing structures.

2 Formulation of the Proposed Algorithm

In this Section, we decompose the computation of the N-point DCT in to 4 number of matrix–vector multiplications, where each matrix is of size (N/4+1) x (N/4). Using the decomposition scheme we then derive the proposed algorithm for high-throughput systolic implementation.

T. Srikanthan et al. (Eds.): ACSAC 2005, LNCS 3740, pp. 787 – 794, 2005.

The DCT of a sequence $\{x(n)$, for $n = 0, 1, 2, \ldots, N\text{-}1\}$ is given by

$$X(k) = (2/N)e(k)\sum_{n=0}^{N-1} x(n)\cos[\alpha_N(2n+1)k], \tag{1}$$

For $0 \le k \le N-1$, and $\alpha_N = \dfrac{\pi}{2N}$, where $e(0) = (1/\sqrt{2})$ and $e(k) = 1$ for $1 \le k \le N-1$. The DCT of (1) can also be expressed as

$$X(k) = A(k)\cos[\alpha_N k] - B(k)\sin[\alpha_N k], . \tag{2}$$

$$A(k) = \sum_{n=0}^{N-1} y(n)\cos[4\alpha_N kn], \tag{3a}$$

and

$$B(k) = \sum_{n=0}^{N-1} y(n)\sin[4\alpha_N kn]. \tag{3b}$$

for $0 \le k \le N-1$, where

$$y(n) = x(2n) \text{ for } 0 \le n \le (N/2)-1, \tag{4a}$$

and

$$y(n) = x(2N-2n-1) \text{ for } (N/2) \le n \le N-1, \tag{4b}$$

$A(k)$ and $B(k)$ in (3) are, respectively, referred to as the even and the odd transform in the rest of the paper. When N is even, one can reduce (3) to sum of $\{(N/2)+1\}$ terms as:

$$A(k) = \sum_{n=0}^{N/2} a(n)\cos[4\alpha_N kn], \tag{5a}$$

And

$$B(k) = \sum_{n=0}^{N/2} b(n)\sin[4\alpha_N kn], \tag{5b}$$

for $0 \le k \le N-1$, where

$$a(n) = y(n) + y(N-n) \tag{6a}$$

$$b(n) = y(n) - y(N-n). \tag{6b}$$

for $1 \le n \le (N/2)-1$.

$$a(0) = y(0)\,, b(0) = 0\,,\ a(N/2) = y(N/2)\ \text{and}\ b(N/2) = 0 \tag{6c}$$

When (N/2) is even (*i.e.*, for $N = 4M$, where M is any positive integer), (5) can be further split in to two halves as

$$A(k) = A_1(k) + A_2(k), \tag{7a}$$

and

$$B(k) = B_1(k) + B_2(k). \tag{7b}$$

for $0 \le k \le N-1$, where

$$A_1(k) = (-1)^k\, y(N/2) + \overline{A_1(k)} \tag{8a}$$

and

$$\overline{A_1(k)} = \sum_{n=0}^{M-1} a_1(n)\cos[2\alpha_M kn]. \tag{8b}$$

$$A_2(k) = \sum_{n=0}^{M-1} a_2(n)\cos[\alpha_M k(2n+1)] \tag{8c}$$

$$B_1(k) = \sum_{n=0}^{M-1} b_1(n)\sin[2\alpha_M kn] \tag{8d}$$

$$B_2(k) = \sum_{n=0}^{M-1} b_2(n)\sin[\alpha_M k(2n+1)] \tag{8e}$$

for $0 \le k \le N-1$.

$$a_1(n) = a(2n),\ a_2(n) = a(2n+1),\ b_1(n) = b(2n),\ \text{and}\ \ b_2(n) = b(2n+1) \tag{9a}$$

for $1 \le n \le N/4-1$, and

$$a_1(0) = y(0),\ a_2(0) = y(1) + y(N-1),\ b_1(0) = 0\ \text{and}\ b_2(0) = y(1) - y(N-1) \tag{9b}$$

Using (7) and (8), on (2) we can express the N-point DCT of $\{x(n)\}$ as

$$X(k) = \big(A_1(k) + A_2(k)\big)\cos[\alpha_N k] - \big(B_1(k) + B_2(k)\big)\sin[\alpha_N k]\,, \tag{10a}$$

$$X(N/2+k) = \big(A_1(k) - A_2(k)\big)\cos[\alpha_N (N/2+k)] - \big(B_1(k) - B_2(k)\big)\sin[\alpha_N (N/2+k)]\,, \tag{10b}$$

for $0 \le k \le (N/4)$.

$$X(N/2-k) = \big(A_1(k) - A_2(k)\big)\sin[\alpha_N(N/2+k)] - \big(B_1(k) + B_2(k)\big)\cos[\alpha_N(N/2+k)], \quad (10c)$$

and

$$X(N-k) = \big(A_1(k) + A_2(k)\big)\sin[\alpha_N k] + \big(B_1(k) + B_2(k)\big)\cos[\alpha_N k] \quad (10d)$$

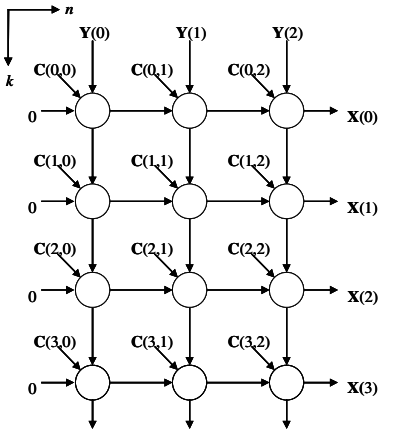
for $0 \le k \le (N/4)$.

From (10) it is clear that four components of the DCT can be computed by a set of 4 components of $A_1(k)$, $B_1(k)$, $A_2(k)$, and $B_2(k)$. The computational complexity is thus reduced from the order of $(N)^2$ to the order of $\{(N)^2\}/4$ since the computation of the DCT now amounts to computation of 4 matrix vector multiplications where each matrix is of size $(N/4+1)$ x $(N/4)$ corresponding to the computation of $A_1(k)$, $B_1(k)$, $A_2(k)$, and $B_2(k)$ for $0 \le k \le (N/4)$.

3 The Proposed Systolic Structure

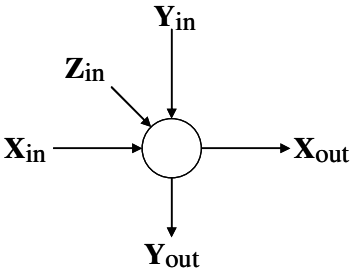
$\overline{A_1(k)}$, $A_2(k)$, $B_1(k)$, and $B_2(k)$ for $0 \le k \le (N/4)$ as given in (8), which comprise the core computation of the DCT can be computed by 4 matrix-vector multiplications, where each matrix is of size $(M+1)$ x M (for $M = N/4$). The dependence graph (DG) in Fig. 1(a) represents the data dependencies and arithmetic-operations for concurrent computation of all the four matrix-vector multiplications for $M=3$. It consists of 4 rows of nodes, where each of the rows consists of 3 nodes. Function of the nodes of the DG is described in Fig. 1 (b). Each node receives three 4-point vector inputs, where the input vector from $[0\ 1]^T$ direction consists of the input sample values and the input vector from $[1\ 1]^T$ direction consists of four coefficient values (one element from each of the four coefficient matrices of size 4 x 3). The input from $[1\ 0]^T$ consists of 4 partial result values. Each node of the right-most column of the DG gives a 4-point vector output as detailed in Fig. 1(a). The DG in Fig. 1 can be projected along k-direction with a default schedule to obtain a linear systolic array of 3 PEs as shown in Fig.2(a). Function of each PE of the systolic array is depicted in Fig. 2(b). During each cycle-period (time-step) each PE receives a set of 4 input samples and a set of 4 coefficients. It performs 4 multiplications as stated in Fig. 2(b), and then adds the product to the partial results available from form it's left. Finally, it sends out a vector consisting of 4 output values to its right across the array in every cycle period. The vertical input to each PE is staggered by one cycle-period with respect to its proceeding PE to satisfy the data-dependence requirement. The structure delivers out the first set of output 3 cycle-periods after the first set of input arrive at the array, and yields the subsequent output in the successive cycle periods. The complete structure for computing the DCT is shown in Fig.3. It requires a post processing unit consisting 8

adder and 8 multipliers to perform the necessary additions, subtractions and multiplications to obtain the DCT components according to (10).



$\mathbf{Y}(n)=[a_1(n) \quad a_2(n) \quad b_1(n) \quad b_2(n)]^{\mathrm{T}}$
for $n=0$, 1and 2.
$\mathbf{X}(k)=[\overline{A_1(k)} \quad A_2(k) \quad B_1(k) \quad B_2(k)]^{\mathrm{T}}$
for $k=0, 1, 2$and 3.
$\mathbf{C}(k,n)=[C_1(k,n) \quad C_2(k,n) \quad C_3(k,n) \quad C_4(k,n)]^{\mathrm{T}}$
$C_1(k,n)=\cos[2\alpha_M kn]$
$C_2(k,n)=\cos[\alpha_M k(2n+1)]$
$C_3(k,n)=\sin[2\alpha_M kn]$
$C_4(k,n)=\sin[\alpha_M k(2n+1)]$
for $k=0,\ 1, 2$ and 3
and for $n=0,\ 1$ and 2.

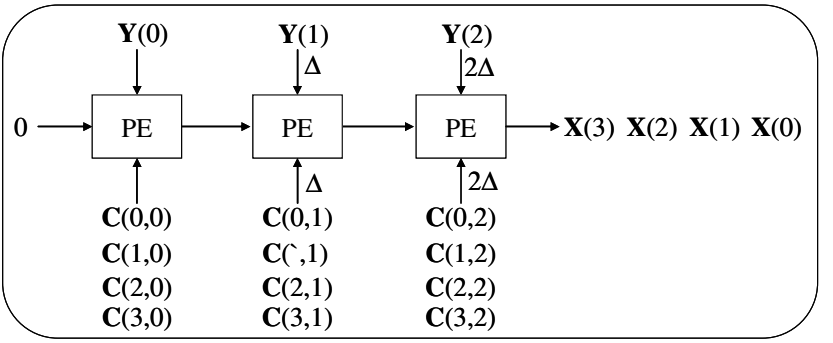
(a)



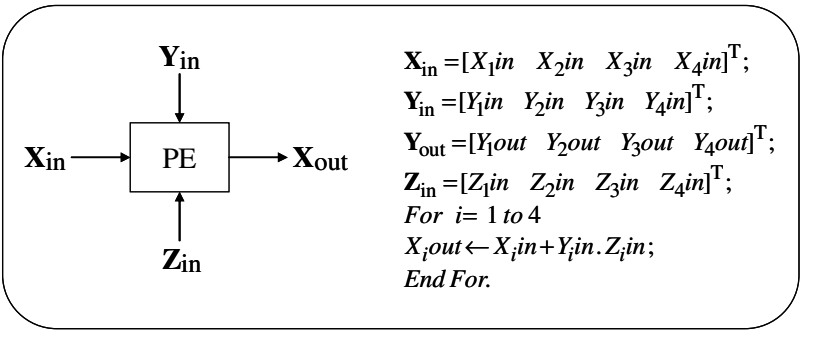
$\mathbf{X}_{\mathrm{in}}=[X_1in \quad X_2in \quad X_3in \quad X_4in]^{\mathrm{T}};$
$\mathbf{X}_{\mathrm{out}}=[X_1out \quad X_2out \quad X_3out \quad X_4out]^{\mathrm{T}};$
$\mathbf{Y}_{\mathrm{in}}=[Y_1in \quad Y_2in \quad Y_3in \quad Y_4in]^{\mathrm{T}};$
$\mathbf{Y}_{\mathrm{out}}=[Y_1out \quad Y_2out \quad Y_3out \quad Y_4out]^{\mathrm{T}};$
$\mathbf{Z}_{\mathrm{in}}=[Z_1in \quad Z_2in \quad Z_3in \quad Z_4in]^{\mathrm{T}};$
For $i=1$ *to* 4
$Y_iout \leftarrow Y_iin;$
$X_iout \leftarrow X_iin+Y_iin.Z_iin;$
End For.

(b)

Fig. 1. The dependence graph for computation of the 4 matrix-vector multiplications for $M = 3$. (a) The DG. (b) Function of each node of the DG.



(a)



(b)

Fig. 2. A linear systolic array for computation of the 4 matrix-vector multiplications for $M = 3$. (a) The linear systolic array. (b) Function of each PE of the systolic array.

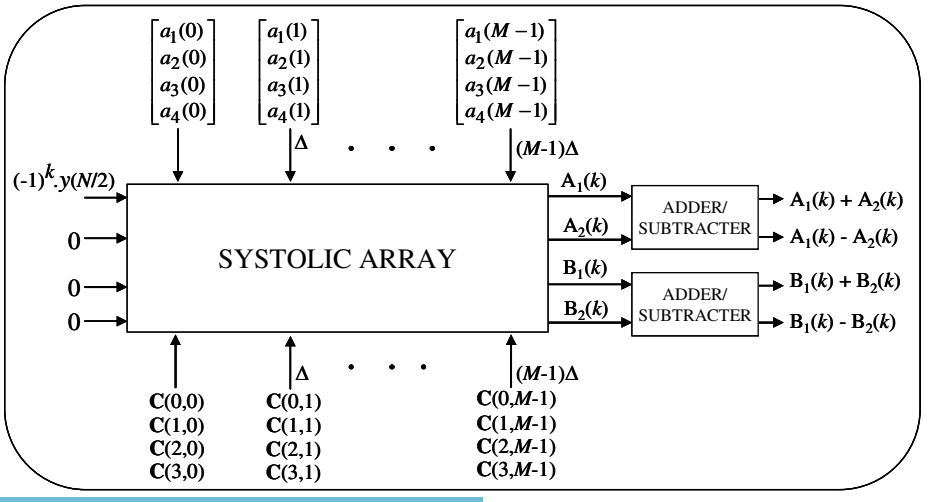


Fig. 3. Proposed structure of computation of the DCT. Δ stands of unit delay.

4 Hardware and Time Complexities

In the previous Section we discussed a systolic structure for computing the DCT for N=4. In general, for computing an N-point transform, the proposed linear systolic array will consist of (N/4) number of PEs (M=N/4), which will yield the first component after (N/4+1) cycle-periods and produces the successive output in every cycle-period thereafter. It would deliver a complete set of N DCT components in every {(N/4)+1} cycle-periods after the pipeline is filled during the first (N/4+1) cycle-periods. During a cycle period each PE performs 4 multiplications and 4 addition operations, where the duration of cycle-period (time-step) depends on the resources available with the PE. If each PE consists of 4 multipliers and 4 adders, then the cycle period T= T_M+T_A , where T_M, and T_A are, respectively, the time required to perform a multiplication and an addition in a PE. The proposed structure for computation of the DCT will, thus, require (N+8) multipliers and (N+8) number of adders including the post-processing unit. The hardware- and time-complexities of the proposed structure are listed along with those of the existing structures [4-9] in Table 1. It is observed that the proposed structure involves a few more multipliers and adders over most of the existing structures, but it offers significantly low latency, very high throughput, and very low area-time complexity compared with the others.

Table 1. Comparison of the Hardware- and Time Complexities of the Proposed Structure and the Existing Structures

Structures	MULT	ADD	CYCLE PERIOD (T)	LATENCY	THROUGHPUT	AREA-TIME COMPLEXITY
Yang [4]	2N-2	3N+2	T_M+2T_A	2NT	1/N	$\approx 2N^3$
Fang [6]	(N/2)+3	N+3	T_M+2T_A	(7/2)NT	1/N	$\approx(N^3)/2$
Guo [7]	(N-1)/2	(N-1)/2	T_M+T_A	(3N-2)T	1/(N-1)	$\approx(N^3)/2$
Pan [5]	N	2N	T_M+T_A	NT	2/N	$\approx(N^3)/4$
Chiper [8]	N-1	N+1	T_M+T_A	2(N-1)T	2/(N-1)	$\approx(N^3)/4$
Cheng [9]	(N-1)/2	3(N-1)/2	T_M+3T_A	2(N-1)T	2/(N-1)	$\approx(N^3)/8$
Proposed	N+8	N+8	T_M+T_A	2(N/4+1)T	1/{(N/4)+1}	$\approx(N^3)/16$

It is found that amongst the existing structures, the structures of Chiper [8] and Pan [5] offer the best throughput while the structure of Cheng [9] offers the least area-time complexity. Compared with the structure of [5], the proposed structure involves 8 more multipliers and (N-8) less adders, while compared with the structure of [8] it involves 9 more multipliers and 7 more adders. But, it offers nearly twice the throughput, and nearly (1/4) times (AT^2) complexity compared with the structures of [5]

and [8] with one-fourth and half the latency of [8] and [5], respectively. The proposed structure requires nearly double the number of multipliers and (2/3)rd the number of adders compared with the structure of [9] but it offers more than double throughput, less than one-fourth latency and less than half area-time complexity of the other.

5 Conclusion

An efficient scheme for decomposition of the DCT into a low-complexity computation-core is derived. A systolic structure is also designed further for the proposed computation-core. The proposed structure is found to have more than twice of the best throughput and nearly half the minimum latency reported so far. The area-time complexity of the proposed structure is also found to be less than half of the best of the existing structures. Apart from that, unlike the existing structures based on circular convolution, it does not require any complex circuits for input and output mapping. It has no restriction on the transform-length and can be scalable for higher transform-length as well. Although the structure is derived for N=4M, where M is any positive integer, it can also be used for any value of N by suitable modification of the upper limit of the indices of summations. The present approach for 1-dimensional DCT can be extended further for computation of 2-dimensional DCT, and it can be extended further for computing the discrete Fourier transform, discrete sine transform, and lapped orthogonal transforms also.

References

1. Natarajan, Ahmed, T. and Rao K. R.: Discrete cosine transform, IEEE Trans. Comput., vol. C-23. (1974) 90–94
2. Kung, S. Y.: VLSI Array Processors. Englewood Cliffs, NJ: Prentice-Hall, (1988)
3. Kung, H. T. : Why systolic architectures?, Comput. Mag, vol. 15. (1982) 37-45
4. Yang, J. F. and Fang, C.-P.: Compact recursive structures for discrete cosine transform, IEEE Trans. Circuits Syst. II, vol. 47. (2000) 314–321
5. Pan, S. B. and Park, R. H.: Unified systolic arrays for computation of the DCT/DST/DHT, IEEE Trans. Circuits Syst. Video Technol., vol. 7. (1997) 413-419
6. Fang, W.-H. and Wu, M.-L.: An efficient unified systolic architecture for the computation of discrete trigonometric transforms, Proc. IEEE Symp. Circuits and Systems, vol. 3. (1997) 2092–2095
7. Guo, J., Liu, C. M. and Jen, C. W.: A new array architecture for prime-length discrete cosine transform, IEEE Trans. Signal Processing, vol. 41. (1993) 436-442
8. Chiper, D. F., Swamy, M. N. S., Ahmad, M. O. and Stouraitis, T.: A systolic array architecture for the discrete sine transform, IEEE Trans. Signal Processing, vol. 50.(2002) 2347-2354
9. Cheng C. and Parhi, K. K.: A Novel Systolic Array Structure for DCT, IEEE Trans. Circuits and Systems II: Express Briefs, Accepted for future publication, 2005.

Efficient VLSI Architectures for Convolution and Lifting Based 2-D Discrete Wavelet Transform

Gab Cheon Jung[1], Seong Mo Park[2], and Jung Hyoun Kim[3]

[1] Dept. of Electronics Engineering, Chonnam National University,
300 Youngbong-Dong, Puk-Gu, Gwangju 500-757, Korea
gcjung@ciscom.chonnam.ac.kr
[2] Dept. of Computer Engineering, Chonnam National University,
300 Youngbong-Dong, Puk-Gu, Gwangju 500-757, Korea
smpark@chonnam.ac.kr
[3] Dept. of Electrical Engineering, North Carolina A&T State University,
1601 East Market Street, Greensboro, NC 27411, USA
kim@ncat.edu

Abstract. This paper presents efficient VLSI architectures for real time processing of separable convolution and lifting based 2-D discrete wavelet transform (DWT). Convolution based architecture uses partitioning algorithm based on the state space representation method and lifting based architecture applies pipelining to each lifting step. Both architectures use recursive pyramid algorithm(RPA) scheduling that intersperses both the row and column operations of the second and following levels among column operations of the first level without using additional filter for row operations of the second and following levels. As a result, proposed architectures have smaller hardware complexity compared to that of other conventional separable architectures with comparable throughput.

1 Introduction

Having studied the method of replacing local Fourier analysis in geophysical signal processing, discrete wavelet transform(DWT) is widely utilized in digital signal fields these days. The wavelet transform not only provides high resolution in time and frequency but also has merit of representing images similar to human optical characteristics. However, since the DWT is implemented using filter banks, it requires extensive operations. To reduce these extensive operations, a lifting scheme was proposed[1] and has been used widely. Nevertheless, a dedicated hardware is indispensable in such a 2-D DWT case owing to huge operations both in row and column directions.

In order to meet the computational requirements for real time processing of 2-D DWT, many VLSI architectures have been proposed and implemented[2-9]. For convolution based approach, Parhi and Nishitani[2] proposed two architectures which combine word-parallel and digit-serial methodologies. Vishwansth et al. [3] presented a systolic-parallel architecture. Chakrabarti and Mumford [4] presented the folded architecture which consists of two parallel computation units and two storage units.

This work was supported by the RRC-HECS, CNU under R0202. The support of IDEC CAD tools and equipment in this research is also gratefully acknowledged.

T. Srikanthan et al. (Eds.): ACSAC 2005, LNCS 3740, pp. 795–804, 2005.

Chakrabarti and Vishwanath[5] proposed the parallel filter architectures for 2-D non-separable DWT. Yu and Chen[6] developed a parallel-systolic filter structure to improve hardware complexity of [5].

Convolution based architectures have easy scalability according to filter length but they generally require larger amount of hardware. On the other hands, lifting based architectures have a difficulty of scaling the structure but have smaller hardware amount compare to convolution based architectures. For lifting based approach, Andra et al. [7] and G. Dillen et al.[8] presented architectures which conduct DWT operations in level by level mode. They require an external RAM module with size of $N^2/4$ for N×N image. Huang et al. [9] presented architecture which implements 2-D lifting based DWT only with line memories by using recursive pyramid algorithm(RPA)[10].

In this paper, we proposed efficient line based VLSI architectures for separable convolution and lifting based 2-D DWT. The rest of the paper is organized as follows. Algorithm decomposition of 2-D DWT is presented in section 2. Section 3 presents the proposed architectures. Comparison of various DWT architectures is described in section 4. Finally, concluding remarks are summarized in Section 5.

2 Algorithm Decomposition

2.1 Convolution Based Algorithm

A separable 2-D DWT can be seen as a 1-D wavelet transform along the rows and a 1-D wavelet transform along the columns[11]. Thus, 2-D DWT can be computed in cascade by filtering rows and columns of an image with 1-D filters. Fig. 1 illustrates a decomposition algorithm of 2-D DWT, where g represents a high pass filter and h represents a low pass filter. At the first level of decomposition, input image is decomposed into two subbands(H, L) by filtering along the rows and H, L subbands are decomposed again into four subbands(HH, HL, LH, LL) by filtering along the columns. The multi-level decomposition is performed with LL subband instead of input image.

In fact, this algorithm is the same as a separable conjugate mirror filter decomposition, and can be viewed as the pyramid structure for 2-D product separable filter banks.

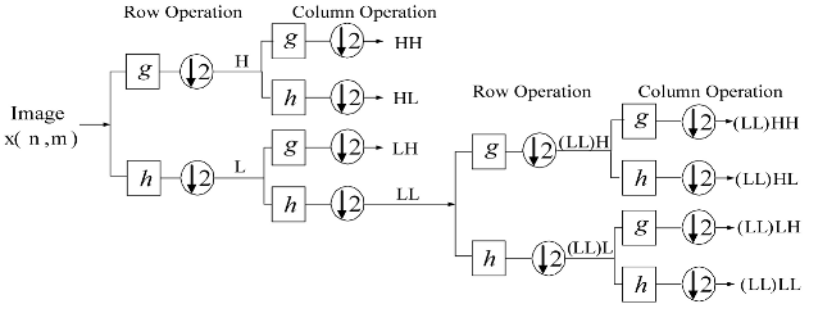


Fig. 1. Decomposition algorithm for 2-D DWT

A 2-D product separable filter can be represented as follows:

$$\begin{aligned} w(n,m) &= \sum_{i=0}^{L_2-1}\sum_{j=0}^{L_1-1} a(i,j)x(n-i,m-j) \\ &= \sum_{j=0}^{L_2-1} a(j) \sum_{i=0}^{L_1-1} a(i)x(n-i,m-j), \end{aligned} \tag{1}$$

where a(i,j) is an impulse response of the product separable 2-D filter and L_1 and L_2 are length of the filter in the horizontal and vertical direction respectively. The second equation of equation(1) indicates a row-column decomposition for 2-D filtering.

The 2-D DWT can be computed by decomposing equation(1) into a set of state space equations using horizontal state variables (q_H^1, q_H^2,..., q_H^k) and vertical state variables (q_V^1, q_V^2,..., q_V^k) in consideration of 2:1 decimation operation as follows:

$$\begin{aligned} &y(n,m)=a(0)x(2n,m)+a(1)x(2n-1,m)+q_H^1(n-1,m), \\ &q_H^{k_1}(n,m)=a(2k_1)x(2n,m)+a(2k_1+1)x(2n-1,m)+q_H^{k_1+1}(n-1,m), \\ &\quad H(n,m)= y(n,m) \text{ for } a(L_1)= g(L_1), \\ &\quad L(n,m)= y(n,m) \text{ for } a(L_1) = h(L_1), \\ &\text{for } n = 0,1,\ldots,(N/2)-1,\ m= 0,1,\ldots,N-1,\ k_1 = 1,2,\ldots,(\lceil L_1/2 \rceil -1). \end{aligned} \tag{2}$$

$$\begin{aligned} &w(n,m)=a(0)y(n,2m)+a(1)y(n,2m-1)+q_V^1(n,m-1), \\ &q_V^{k_2}(n,m)=a(2k_2)y(n,2m)+a(2k_2+1)y(n,2m-1)+q_V^{k_2+1}(n,m-1), \\ &\quad HH(n,m)=w(n,m) \text{ for } a(L_2)= g(L_2) \text{ and } y(n,m)=H(n,m), \\ &\quad HL(n,m)=w(n,m) \text{ for } a(L_2)= h(L_2) \text{ and } y(n,m)=H(n,m), \\ &\quad LH(n,m)=w(n,m) \text{ for } a(L_2)= g(L_2) \text{ and } y(n,m)=L(n,m), \\ &\quad LL(n,m)=w(n,m) \text{ for } a(L_2)= h(L_2) \text{ and } y(n,m)=L(n,m), \\ &\text{for } n= 0,1,\ldots,(N/2)-1,\ m = 0,1,\ldots,(N/2)-1,\ k_2 = 1,2,\ldots,(\lceil L_2/2 \rceil -1). \end{aligned} \tag{3}$$

Equation(2) represents row operations in horizontal direction and uses inputs x(2n,m), x(2n-1,m) and previous horizontal state variables q_H(n-1,m) for the computation, where a(0), a(1), a(2),...,a(L_1-1) denote high pass filter coefficients($g(0)$, $g(1)$,...,$g(L_1-1)$) or low pass filter coefficients ($h(0)$, $h(1)$,...,$h(L_1-1)$). Equation (3) represents column operations in vertical direction and uses results (y(n,2m), y(n,2m-1)) of equation (2) and previous vertical state variables q_V(n,m-1). In equation(2)and(3), each output and state variable uses two inputs and one previous state variable and requires two multiplications and three additions. Thus, it can be computed with critical path having 1 multiplication and 2 additions regardless of wavelet filer length.

2.2 Lifting Scheme

Daubeches and Sweldens[1] proposed the lifting scheme in order to reduce extensive operations of filter bank. The lifting scheme decomposes every DWT operations into finite sequences of simple filtering steps. The lifting steps are consists of three

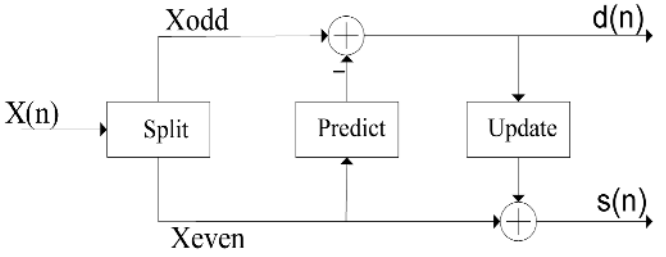


Fig. 2. Lifting Step

steps(splitting, prediction, and update) as shown in fig. 2. In splitting step, input signals are split into even samples and odd samples. The prediction step predicts odd samples from even samples and computes high pass coefficients(d) by calculating the difference between the odd samples and the prediction values. In update step, the low pass coefficients(s) are computed from high pass coefficients.

The lifting scheme has many advantages such as a fast computation, in place calculation, integer-to-integer, easiness of inverse implementation, etc. Because of these merits, the JPEG2000 standard supports lifting scheme as well as convolution method for DWT operations. It adopts integer (5,3) filter for reversible transform and biorthogonal (9,7) filter for irreversible transform as default mode[12,13]. While high pass and low pass results for (5,3) filter are computed by predict and update step once, results for (9,7) filter require one more predict and update step.

The lifting equations for biorthogonal (9,7)/(5,3) filter of the separable 2-D DWT can be represented by equation(4) and equation(5) where K is a scaling factor and is 1.0 for (5,3) filter. Each predict step with filter coefficient α, γ produces high pass results. And update steps with filter coefficient β, δ produce low pass results. In this equation, we can consider $s^0(n)$, $d^1(n)$, $s^1(n)$, and $d^2(n)$ inputs of each lifting step to be state variables since the present state of these inputs are used for next lifting operations.

$$
\begin{aligned}
&d_H^0(n,m)=x(2n+1,m),\\
&s_H^0(n,m)=x(2n,m),\\
&d_H^1(n,m)=d_H^0(n,m)+\alpha\,(s_H^0(n,m)+s_H^0(n+1,m)),\\
&s_H^1(n,m)=s_H^0(n,m)+\beta\,(d_H^1(n-1,m)+d_H^1(n,m)),\\
&d_H^2(n,m)=d_H^1(n,m)+\gamma\,(s_H^1(n,m)+s_H^1(n+1,m)),\\
&s_H^2(n,m)=s_H^1(n,m)+\delta\,(d_H^2(n-1,m)+d_H^2(n,m)),\\
&\quad H(n,m)=d_H^1(n,m), \qquad L(n,m)=s_H^1(n,m) \qquad \text{for (5,3)filter}\\
&\qquad\quad =(1/K)d_H^2(n,m) \qquad\quad =Ks_H^2(n,m) \qquad \text{for (9,7) filter,}\\
&\text{for } n=0,1,\ldots,(N/2)-1,\ m=0,1,\ldots,N-1.
\end{aligned}
\tag{4}
$$

$$
\begin{aligned}
&d_V^0(n,m)=H(n,2m+1) \text{ or } L(n,2m+1),\\
&s_V^0(n,m)=H(n,2m) \text{ or } L(n,2m),\\
&d_V^1(n,m)=d_V^0(n,m)+\alpha\,(s_V^0(n,m)+s_V^0(n,m+1)),\\
&s_V^1(n,m)=s_V^0(n,m)+\beta\,(d_V^1(n,m-1)+d_V^1(n,m)),\\
&d_V^2(n,m)=d_V^1(n,m)+\gamma\,(s_V^1(n,m)+s_V^1(n,m+1)),\\
&s_V^2(n,m)=s_V^1(n,m)+\delta\,(d_V^2(n,m-1)+d_V^2(n,m)),\\
&\quad HH(n,m)=Hd_V^1(n,m), \qquad HL(n,m)=Hs_V^1(n,m) \qquad \text{for (5,3)filter}\\
&\qquad\quad =H(1/K)d_V^2(n,m) \qquad\quad =H(K)s_V^2(n,m) \qquad \text{for(9,7)filter}\\
&\quad LH(n,m)=Ld_V^1(n,m), \qquad LL(n,m)=Ls_V^1(n,m) \qquad \text{for (5,3)filter}\\
&\qquad\quad =L(1/K)d_V^2(n,m) \qquad\quad =L(K)s_V^2(n,m) \qquad \text{for(9,7)filter,}\\
&\text{for n= 0,1,. . .,(N/2)-1, m = 0,1,. . .,(N/2)- 1.}
\end{aligned}
\tag{5}
$$

3 Proposed Architectures

This section describes the proposed VLSI architectures for convolution and lifting based 2-D DWT. For consistency in description of our architectures, we use the biorthogonal (9,7) filter. The proposed architecture for 2-D convolution based DWT consists of a horizontal(HOR) filter, signal memory(SIG MEM), and a vertical(VER) filter as shown in fig. 3. The architecture for 2-D lifting based DWT also has the same structure as in fig. 3.

3.1 HOR Filter

The HOR filter conducts DWT operation along the rows with two columns of image from the internal serial-to-parallel converter which splits serial input image data into parallel column data(odd, even) at every other clock cycle. For convolution based architecture, symmetric characteristic of biorthogonal filter coefficients is used. That is, two same multiplications of filter coefficient and input in equation(2) share one multiplier. Thus, the number of multiplier is reduced to a half. The HOR filter for convolution contains 5 multipliers, 8 adders, and 4 S.V(State Variable) registers as shown in fig. 4 and outputs high pass result or low pass result at every clock cycle.

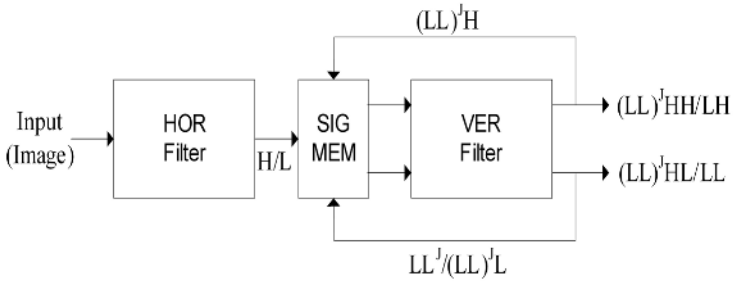


Fig. 3. Block diagram of proposed architecture for 2-D DWT

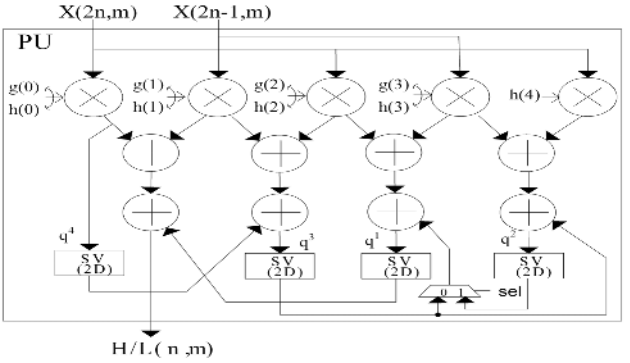


Fig. 4. HOR filter structure: convolution based DWT

The HOR filter for lifting scheme computes equation (4). Fig. 5 shows HOR filter for lifting based DWT. For (9, 7) filter, the PE0 computes predict steps and PE1 computes update steps. The lifting based architecture contains 3 multipliers, 4 adders, 2 S.V(State Variable) registers, 6 registers for pipelining each lifting step.

3.2 Signal Memory

The signal memory is used to store intermediate results and to send these results to the VER filter. It includes odd and even line buffers as shown in fig. 6. Odd line buffer stores one odd row of $(LL)^JH$ and $(LL)^JL$ ($J \geq 0$) subbands or odd column for one row of $(LL)^J$ ($J \geq 1$) subbands. And even line buffer stores even row of $(LL)^JH$ and $(LL)^JL$ ($J \geq 1$) subbands. One register in even line buffer is used to store present output from the HOR filter for even row of image.

3.3 VER Filter

For the operational scheduling in our scheme, the VER filter conducts column operations at the first level when the HOR filter outputs H, L results for even row of image.

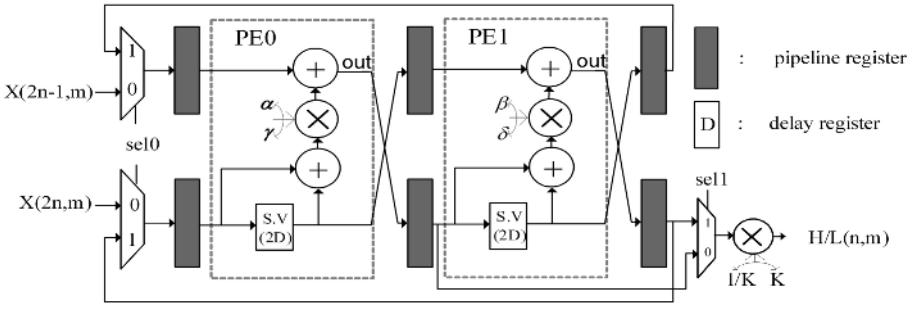


Fig. 5. HOR filter structure: lifting based DWT

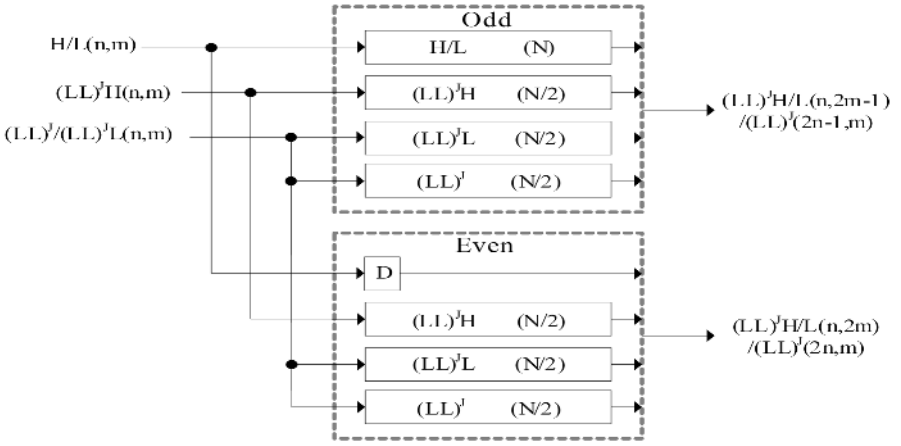


Fig. 6. The signal memory for intermediate results

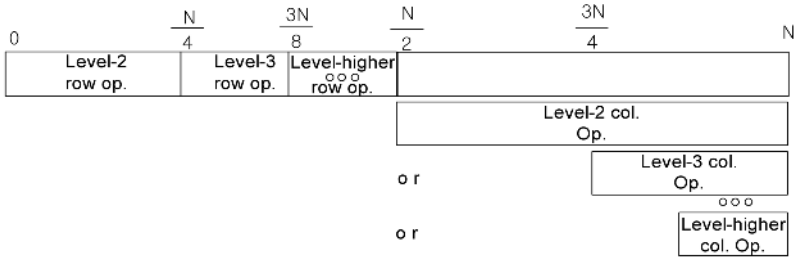


Fig. 7. Timing diagram of VER filter for odd rows

On the other hands, when the HOR filter outputs H, L results for odd row of image, it conducts both row operations and column operations at the second and following levels as showin in fig. 7. These operations were scheduled with consideration of the computing time and data dependency. Since the VER filter computes high pass and low pass outputs concurrently, the computing time of column operation for one single line at the J level(J>0) is $N/(2)^{J-1}$ and the computing time of row operations for single line at the J level(J>1) is $(1/2)\times(N/(2)^{J-1})=N/(2)^{J}$. Fig. 8 shows data flow of the VER filter with signal memory according to above scheduling.

The VER filter for convolution consists of two processing units as shown in fig. 9. It computes equation(2) for row operations with two columns of $(LL)^{J-1}$ (J>1) subbands and equation(3) for column operations with two rows of $H(LL)^{J}$, $L(LL)^{J}$ (J>0) subbands. The VER filter for lifting scheme consists of 4 processing elements as shown in fig. 10 and computes equation(4) for row operations and equation(5) for column operations. Each processing element conducts each lifting step.

The internal structure of each PU and PE in a VER filter is identical to that of HOR filter except for filter coefficients(PU:low-pass, high-pass, PE: PE0-α , PE1-β, PE2-γ, PE3-δ) and state variable(S.V) buffers. Each horizontal state variable(q_H) in VER

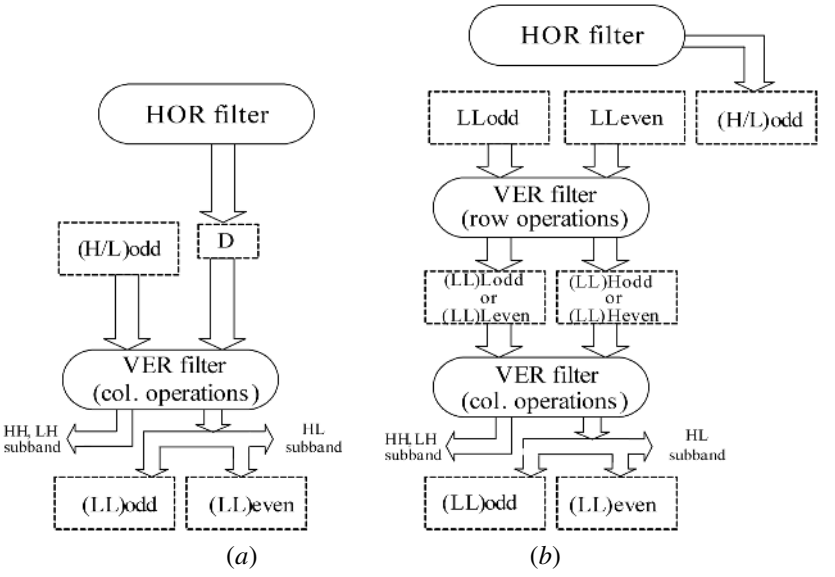


Fig. 8. Data flow of VER filter at (a)even rows and (b)odd rows of image

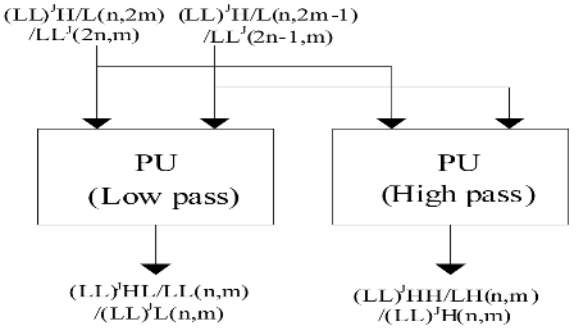


Fig. 9. The VER filter structure: convolution based DWT

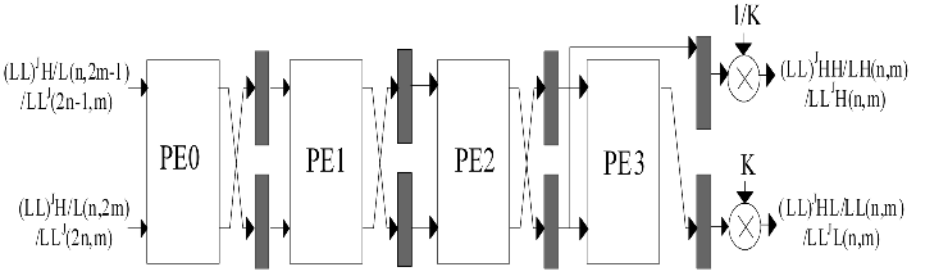


Fig. 10. The VER filter structure: lifting base DWT

filter requires 1 register for row operations. And each vertical state variables for vertical filtering are stored in buffers with size of $2N(N+(N/2)+(N/4)\ldots+N/2^{J-1})$ because vertical state variables for one row at all the decomposition level must be stored for the computation of next row.

4 Performance and Comparisons

While conventional 2-D architectures that implement RPA have 50%~66.7% hardware utilization, proposed architectures have higher hardware utilization. Contrary to the conventional RPA based architectures in [4][5] which have independent filter for row operations of the second and following levels, proposed architectures make the VER filter compute not only all levels along columns but also the second and following levels along rows. This leads that proposed architectures have 66.7%~88.9% hardware utilizations.

Table 1 shows comparisons of proposed architectures and previous conventional 2-D DWT architectures, where Tm is the latency of multiplication and Ta is the latency of addition. Comparison is performed with the number of multipliers, adders, storage size, and critical path. Note that the scaling factors are involved in multipliers and pipelining is not applied to critical path, and symmetric characteristic of biorthogonal filter is considered for convolution based architectures. In table 1, architectures in [4], [5] and proposed architecture(convolution) use convolution method and architectures in [9] and proposed(lifting) use lifting scheme. When compared with architectures in [4], [5], proposed one(convolution) has not only less amount of hardware but can reduce critical path. While architectures in [4], [5] have the critical path having $Tm+\lceil Log_2 L \rceil Ta$, the proposed one(convolution) has critical path having Tm+ 2Ta regardless of filter length. When compare the proposed architecture(lifting) with [9], we know that proposed one(lifting) has less amount of hardware such as multipliers and adders through the improvement of hardware utilization.

5 Conclusion

In this paper, RPA based 2-D DWT architectures for convolution and lifting scheme were described. Proposed architectures do not use additional independent filter for row

Table 1. Comparison of various 2-D architectures for (9,7) filter

	Multipliers	Adders	Storage size	Computing time	Critical path
Proposed (Convolution)	14	22	18N	$\approx N^2$	Tm+2Ta
Systolic-Parallel[5]	19	30	22N	$\approx N^2$	Tm+4Ta
Folded[4]	19	30	18N+(3/2)N	$\approx N^2$	Tm+4Ta
Proposed (Lifting)	9	12	12N	$\approx N^2$	4Tm+8Ta
Lifting[9]	12	16	14N	$\approx N^2$	4Tm+8Ta

operations of the second and following levels. As a result, regarding biorthogonal (9,7) filter, convolution based architecture has 14 multipliers, 22 adders, and 18N line memories for N×N image. And lifting based one requires 9 multipliers, 12 adders, and 12N line memories. With the reduction of hardware cost, convolution based architecture has critical path having 1 multiplication and 2 additions regardless of wavelet filter length by state space representation method. Proposed architectures are very suitable for single chip design due to small hardware cost and regularity.

References

1. I. Daubechies and W. Sweldens.: Factoring wavelet transforms into lifting schemes. The Journal of Fourier Analysis and Applications, Vol. 4, (1998) 247-269
2. K. K . Parhi and T. Nishitani.: VLSI Architectures for Discrete Wavelet Transforms. IEEE Trans. VLSI Systems. Vol. 1, No. 2, (June 1993) 191-202
3. M. Vishwanath, R. M. Owens, and M. J. Irwin.: VLSI architectures for the discrete wavelet transform. IEEE Trans. Circuits and Systems II, Analog and digital Signal Processing, Vol. 42, No. 5, (May 1995) 305-316
4. C. Chakrabarti and C. Mumford.: Efficient Realizations of encoders and decoders based on the 2-D Discrete Wavelet Transform. IEEE Trans. VLSI Systems, (September 1999) 289-298
5. C. Chakrabarti and M. Vishwanath.: Efficient Realizations of the Discrete and Continuous Wavelet Transforms: from Single Chip Implementations to Mappings on SIMD array Computers. IEEE Trans. Signal Processing Vol. 43, No. 3, (Mar. 1995) 759-771
6. Chu Yu and Sao-Jie Chen.: VLSI Implementation of 2-D Discrete Wavelet Transform for Real-Time Video Signal Processing. IEEE Trans. Consumer Electronics, Vol. 43, No. 4, (November 1997) 1270-1279
7. K. Andra, C. Chakrabarti, and T. Acharya.: A VLSI Architecture for Lifting-Based Forward and Inverse Wavelet Transform. IEEE Trans. Signal Processing, Vol. 50, No. 4, (April 2002) 966-977
8. G. Dillen, B. Georis, J.D. Legat, and O. Cantineau.: Combined Line-Based Architecture for the 5-3 and 9-7 Wavelet Transform of JPEG2000. IEEE Trans. Circuits and Systems for Video Technology, Vol. 13, No. 9, (September 2003) 944-950
9. C. T. Huang, P. C. Tseng, and L.G. Chen.: Efficient VLSI architectures of Lifting-Based Discrete Wavelet Transform by Systematic Design Method. IEEE International Symp. on Circuits and Systems, vol. 5, (May 2002) 26-29
10. M. Vishwanath.: The Recursive Pyramid Algorithm for the Discrete Wavelet Transform. IEEE Trans. Signal Processing, Vol.42, No.3, (March 1994) 673-677
11. S. G. Mallet.: A theory of multiresolution signal decomposition : the wavelet representation. IEEE Trans. on Pattern Recognition and Machine Intelligence, Vol. 11, No. 7, (July 1989) 674-693
12. ISO/IEC JTC1/SC29/WG1 (ITU-T SG8) N2165.: JPEG2000 Verification Model 9.1 (Technical Description), (June 2001)
13. C. Christopoulos, A. Skodras and T. Rbrahimi.: The JPEG2000 Still Image coding System: An Overview. IEEE Trans. On Consumer Electronics, Vol. 46, No. 4, (November 2000) 1103-1127

A Novel Reversible TSG Gate and Its Application for Designing Reversible Carry Look-Ahead and Other Adder Architectures

Himanshu Thapliyal and M.B. Srinivas

Center for VLSI and Embedded System Technologies,
International Institute of Information Technology,
Hyderabad-500019, India
thapliyalhimanshu@yahoo.com, srinivas@iiit.net

Abstract. Reversible logic is emerging as a promising area of research having its applications in quantum computing, nanotechnology, and optical computing. The classical set of gates such as AND, OR, and EXOR are not reversible. In this paper, a new 4 *4 reversible gate called "TSG" gate is proposed and is used to design efficient adder units. The proposed gate is used to design ripple carry adder, BCD adder and the carry look-ahead adder. The most significant aspect of the proposed gate is that it can work singly as a reversible full adder i.e reversible full adder can now be implemented with a single gate only. It is demonstrated that the adder architectures using the proposed gate are much better and optimized, compared to their counterparts existing in literature, both in terms of number of reversible gates and the garbage outputs.

1 Introduction

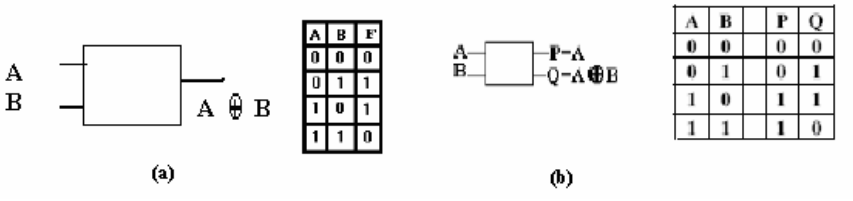
This section provides a simplified description of reversible logic with definitions and motivation behind it.

1.1 Definitions

Researchers like Landauer have shown that for irreversible logic computations, each bit of information lost, generates kTln2 joules of heat energy, where k is Boltzmann's constant and T the absolute temperature at which computation is performed [1]. Bennett showed that kTln2 energy dissipation would not occur, if a computation is carried out in a reversible way [2], since the amount of energy dissipated in a system bears a direct relationship to the number of bits erased during computation. Reversible circuits are those circuits that do not lose information. Reversible computation in a system can be performed only when the system comprises of reversible gates. These circuits can generate unique output vector from each input vector, and vice-versa. In the reversible circuits, there is a one-to-one mapping between input and output vectors. Classical logic gates are irreversible since input vector states cannot be uniquely reconstructed from the output vector states. Let us consider Fig. 1.a from

T. Srikanthan et al. (Eds.): ACSAC 2005, LNCS 3740, pp. 805 – 817, 2005.

which it is obvious that a unique input vector cannot be constructed from the output vector. This is because for output 0, there are two input vectors AB=(00, 11). Now, considering Fig. 1.b, a reversible XOR gate, a unique input vector can be constructed for every output vector.



A	B	F
0	0	0
0	1	1
1	0	1
1	1	0

A	B	P	Q
0	0	0	0
0	1	0	1
1	0	1	1
1	1	1	0

Fig. 1. (a) Classical XOR gate (b) Reversible XOR gate

1.2 Motivation Behind Reversible Logic

According to Laundauer [1,2], the use of logically irreversible gate, dissipates energy into the environment. In other words, Information Loss=Energy Loss. Whenever a logic operation is performed, the computer erases information. Bennett's theorem [2] about heat dissipation is only a necessary and not a sufficient condition, but its extreme importance lies in the fact that every future technology will have to use reversible gates to reduce power. The current technologies employing irreversible logic dissipate a lot of heat and thus reduce the life of a circuit. The reversible logic operations do not erase (lose) information and dissipate very less heat. Thus, reversible logic is likely to be in demand in futuristic high-speed power-aware circuits. Reversible circuits are of high interest in low-power CMOS design [10], optical computing [11], nanotechnology [13] and quantum computing [12]. The most prominent application of reversible logic lies in quantum computers. A quantum computer will be viewed as a quantum network (or a family of quantum networks) composed of quantum logic gates; each gate performing an elementary unitary operation on one, two or more two–state quantum systems called qubits. Each qubit represents an elementary unit of information corresponding to the classical bit values 0 and 1. Any unitary operation is reversible hence quantum networks affecting elementary arithmetic operations such as addition, multiplication and exponentiation cannot be directly deduced from their classical Boolean counterparts (classical logic gates such as AND or OR are clearly irreversible).Thus, Quantum Arithmetic must be built from reversible logical components [14].

1.3 Definition-Garbage Output

Garbage output refers to the output that is not used for further computations. In other words, it is not used as a primary output or as an input to other gate. Figure 2 below elucidates the garbage output marked by *. Minimization of garbage outputs is one of the major challenges in reversible logic.

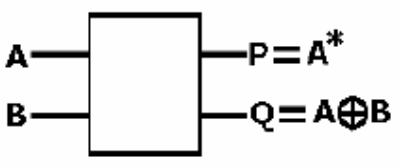


Fig. 2. Illustration of Garbage Output

1.4 A New Reversible TSG Gate

In this paper, the authors propose a new reversible 4*4 TSG gate and use it to design reversible ripple carry adder, BCD adder and carry look-ahead adder circuits. It is shown that the adder architectures using the proposed TSG gate are better than the existing ones in literature, in terms of number of reversible gates and garbage outputs. The reversible circuits designed in this paper form the basis of an ALU of a primitive quantum CPU [17,18,19].

2 Basic Reversible Gates

There are a number of existing reversible gates in literature. Five most important gates are used to construct the adder circuits along with the proposed TSG gate. A brief description of the gates is given below.

2.1 Fredkin Gate

Fredkin gate [3], a (3*3) conservative reversible gate originally introduced by Petri [4,5]. It is called 3*3 gate because it has three inputs and three outputs. The term conservative means that the Hamming weight (number of logical ones) of its input equals the Hamming weight of its output. The input triple (x_1,x_2,x_3) associates with its output triple (y_1, y_2,y_3) as follows:

$$y1=x1$$
$$y2= \sim x1x2+x1x3$$
$$y3=x1x2+ \sim x1x3$$

Fredkin gate behaves as a conditional switch, that is, FG(1; x2; x3) = (1; x3; x2) and FG(0; x2; x3) = (0; x2; x3). Figure 3 shows Fredkin gate symbol and its operation as a conditional switch.

2.2 New Gate

New Gate (NG) [6] is a 3*3 one-through reversible gate as shown in Fig. 4. The input triple (A,B,C) associates with its output triple(P,Q,R) as follows:

P=A; Q=AB^C; R=A'C'^B';

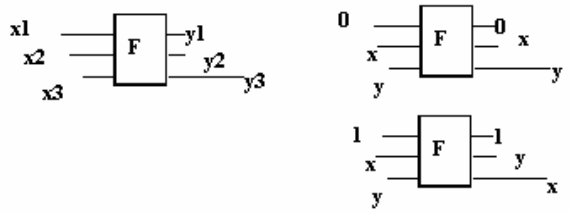


Fig. 3. Fredkin Gate Symbol and Its working as a Conditional Switch

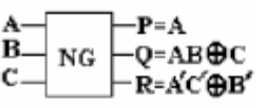


Fig. 4. New Gate

2.3 R2 Gate

The R2 gate is a 4*4 reversible gate proposed in [20]. Figure 5 shows R2 gate along with input and output mapping.

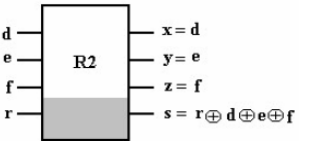


Fig. 5. R2 Gate

2.4 Feynman Gate

Feynman gate [3,7,8] is a 2*2 one-through reversible gate shown in Fig.6. It is called 2*2 gate because it has 2 inputs and 2 outputs. One through gate means that one input variable is also the output. The input double (x1,x2) associates with its output double (y1,y2) as follows.

Y1=x1;
Y2=x1^x2;

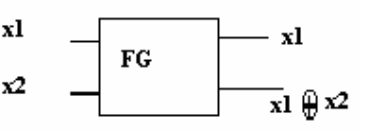


Fig. 6. Feynman Gate

3 Proposed 4* 4 Reversible Gate

In this paper, a '4*4 one through' reversible gate called TS gate "TSG" is proposed and is shown in Fig. 7.The corresponding truth table of the gate is shown in Table 1. It can be verified from the truth table that the input pattern corresponding to a particular output pattern can be uniquely determined. The proposed TSG gate can implement all Boolean functions. Figure 8 shows the implementation of reversible XOR gate using the proposed gate and also the implementation of reversible NOT and NOR functions. Since, NOR gate is a universal gate and any Boolean function can be implemented through it, it follows that proposed gate can also be used to implement any Boolean function.

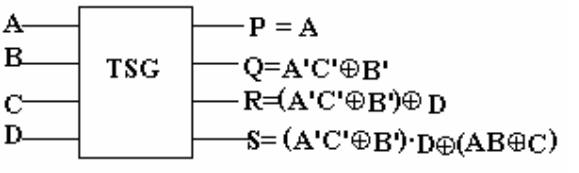


Fig. 7. Proposed 4 * 4 TSG Gate

Table 1. Truth Table of the Proposed TSG Gate

A	B	C	D	P	Q	R	S
0	0	0	0	0	0	0	0
0	0	0	1	0	0	1	0
0	0	1	0	0	1	1	1
0	0	1	1	0	1	0	0
0	1	0	0	0	1	1	0
0	1	0	1	0	1	0	1
0	1	1	0	0	0	0	1
0	1	1	1	0	0	1	1
1	0	0	0	1	1	1	0
1	0	0	1	1	1	0	1
1	0	1	0	1	1	1	1
1	0	1	1	1	1	0	0
1	1	0	0	1	0	0	1
1	1	0	1	1	0	1	1
1	1	1	0	1	0	0	0
1	1	1	1	1	0	1	0

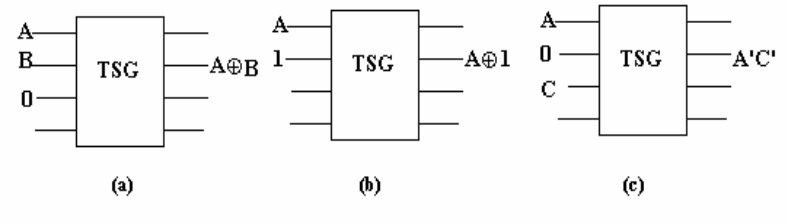


Fig. 8. (a) TSG Gate as XOR Gate (b) TSG Gate as NOT Gate (c) TSG Gate as NOR Gate

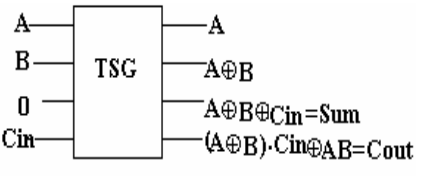


Fig. 9. TSG Gate as Reversible Full Adder

One of the most prominent features of the proposed gate is that it can work singly as a reversible full adder unit. Figure 9 shows the implementation of the proposed gate as a reversible full adder. A number of reversible full adders have been proposed in literature, for example, in [6,7,8,9]. While the reversible full adder circuit in [6] requires three reversible gates (two 3*3 new gate and one 2*2 Feynman gate) and produces three garbage outputs, the same in [7,8] requires three reversible gates (one 3*3 new gate, one 3*3 Toffoli gate and one 2*2 Feynman gate) and produces two garbage outputs. The design in [9] requires five reversible Fredkin gates and produces five garbage outputs. The proposed full adder using TSG in Fig. 9 requires only one reversible gate, that is, one TSG gate and produces only two garbage outputs. Hence, the full-adder design in Fig. 9 using TSG gate can be considered more optimal than the previous full-adder designs of [6,7,8,9]. A comparison of results is shown in Table 2.

Table 2. Comparitive Results of different full adder circuits

	Number of Reversible Gates Used	Number of Garbage Outputs Produced	Unit Delay
Proposed Circuit	1	2	1
Existing Circuit[6]	3	3	3
ExistingCircuit [7,8]	3	2	2
Existing Circuit[9]	5	5	5

4 Applications of the Proposed TSG Gate

To illustrate the application of the proposed TSG gate three different types of reversible adders – ripple carry adder, BCD (Binary Code Decimal) adder and carry look-ahead adders are designed. It is shown that the adder circuits derived using the proposed gate are the most optimized ones compared to others mentioned above.

4.1 Ripple Carry Adder

The full adder is the basic building block in a ripple carry adder. The reversible ripple carry adder using the proposed TSG gate is shown in Fig. 10 which is obtained by cascading the full adders in series. The output expressions for a ripple carry adder are:

$$S_i = A \hat{} B \hat{} Ci;$$
$$C_{i+1} = (A \hat{} B).Ci \hat{} AB \ (i=0, 1, 2\ldots.)$$

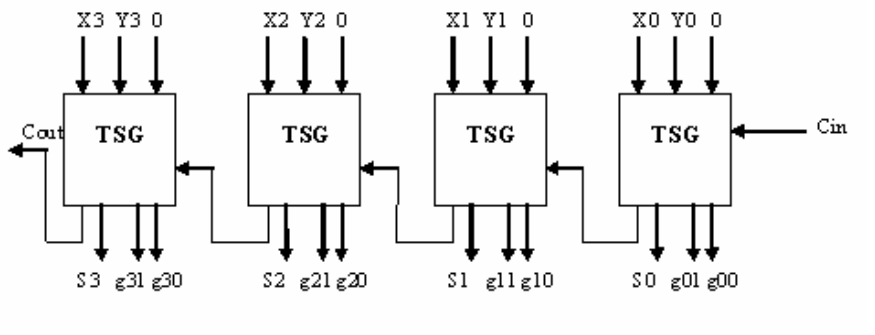


Fig. 10. Ripple Carry Adder Using the Proposed TSG Gate

Evaluation of the Proposed Ripple Carry Adder. It can be inferred from Fig. 10 that for N bit addition; the proposed ripple carry adder architecture uses only N reversible gates and produces only 2N garbage outputs. Table 3 shows the results that compare the proposed ripple carry adder using TSG gate, with those designed using existing full adders of [6,7,8,9]. It is observed that the proposed circuit is better than existing ones both in terms of number of reversible gates and garbage outputs.

4.2 BCD (Binary Coded Decimal) Adder

A BCD adder is a circuit that adds two BCD digits in parallel and produces a sum digit also in BCD. Figure 11 shows the conventional BCD adder. A BCD adder must also include the correction logic in its internal construction. The two decimal digits, together with the input carry, are first added in the top 4-bit binary adder to produce the binary sum. When the output carry is equal to zero, nothing is added to

the binary sum. When it is equal to one, binary 0110 is added to the binary sum using another 4-bit binary adder (bottom binary adder). The output carry generated from the bottom binary adder can be ignored. Fig. 12 shows the proposed reversible BCD adder using the proposed reversible TSG gate. For optimized implementation of the BCD adder; New Gate (NG) is also used for producing the best optimized BCD adder.

Table 3. Comparative Results of the Reversible Ripple Carry Adders

	Number of Reversible Gates Used	Number of Garbage Outputs Produced	Unit Delay
Proposed Circuit	N	2N	N
Existing Circuit [6]	3N	3N	3N
Existing Circuit[7,8]	3N	2N	3N
Existing Circuit[9]	5N	5N	5N

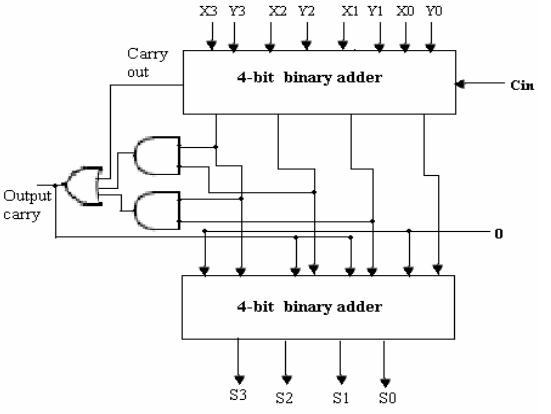


Fig. 11. Conventional BCD Adder

Evaluation of the Proposed BCD Adder. The proposed 4-bit BCD adder architecture in Fig. 12 using TSG gates uses only 11 reversible gates and produces only 22 garbage outputs. A reversible BCD adder has been proposed recently [15] but, the BCD adder the proposed TSG gate is better than the architecture proposed in [15], both in terms of number of reversible gates used and garbage output produced. Table 4 shows the results that compare the proposed BCD carry adder using TSG gate with the BCD adder proposed in [15].

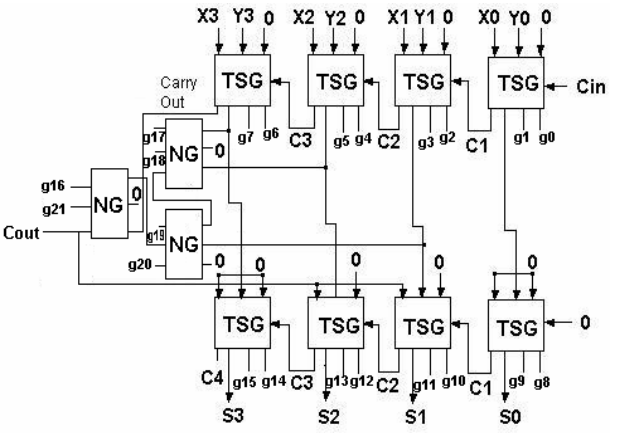


Fig. 12. Proposed Reversible BCD Adder Using TSG Gate

Table 4. Comparative Results of Reversible BCD Adders

	Number of Reversible Gates Used	Number of Garbage Outputs Produced
Existing Reversible BCD Adder[15]	23	22
Proposed Reversible BCD Adder	11	22

4.3 Carry Look Ahead Adder

Carry look-ahead adder is the most widely used technique employed for reducing the propagation delay in a parallel adder. In carry look-ahead adder, we consider two new binary variables:

$$Pi=Ai\hat{}Bi$$
$$Gi=AiBi$$

Then, the output sum and carry of full adder can be rewritten as follows:

$$Si=Ai\hat{}Bi\hat{}Ci=Pi\hat{}Ci$$
$$Ci+1=AiBi+(Ai\hat{}Bi)Ci=AiBi\hat{}\ (Ai\hat{}Bi)Ci=Gi\hat{}PiCi$$

The 4 bit reversible Carry look-ahead adder is created by expanding the above equations.

$$C1=G0\text{^}P0C0$$
$$C2=G1\text{^}P1C1=G1+P1G0+P1P0C0$$
$$C3=G2\text{^}P2C2=G2+P2G1+P2P1G0+P2P1P0C0$$

In the proposed 4-bit reversible carry look-ahead adder, the Pi, Gi and Si are generated from reversible TSG gate and Fredkin gate. The block generating Pi, Gi and Si is called PGA (Carry Propagate & Carry Generate Adder). The structure of PGA is shown in Fig. 13 and the complete structure of the 4-bit carry look-ahead adder is shown in Fig.14. In the complete design of CLA in Fig.14 appropriate gates are used wherever required for generating the function with minimum number of reversible gates and garbage outputs. The unused outputs in the Fig.14 represent the garbage outputs. In the design of a reversible circuit, copies of some of the bits are needed for computation purpose leading to fan-out which is strictly prohibited in reversible computation. In such a case, a reversible gate should be used for copying, but inappropriate selection of gate can lead to generation of garbage outputs. Thus, Feynman gate is used in this work for copying the bits. As there are exactly two outputs corresponding to the inputs of a Feynman gate, a '0' in the second input will copy the first input in both the outputs of that gate. So, Feynman gate is the most suitable gate for single copy of bit, since it does not produce any garbage output.

Evaluation of the Proposed CLA. The proposed 4-bit reversible carry look-ahead adder is the most optimal compared to its existing counterparts in literature. The earlier proposed carry look-ahead adder in [16] has 22 gates while the proposed carry look-ahead adder has only 19 gates. Furthermore, the existing carry look ahead in literature uses two 5*5 reversible C5Not gate while the proposed reversible Carry look Ahead adder does not use any 5*5 reversible gate. Hence, the proposed reversible carry look-ahead adder is also optimized in terms of garbage outputs and number of reversible gate used. Table 5 provides a comparison of results of the reversible carry look-ahead adders.

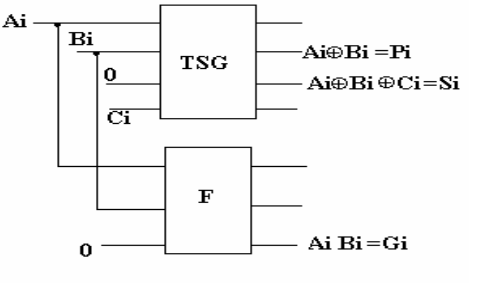


Fig. 13. PGA Block in CLA generating Sum, Pi and Gi Using TSG and Fredkin Gate(F)

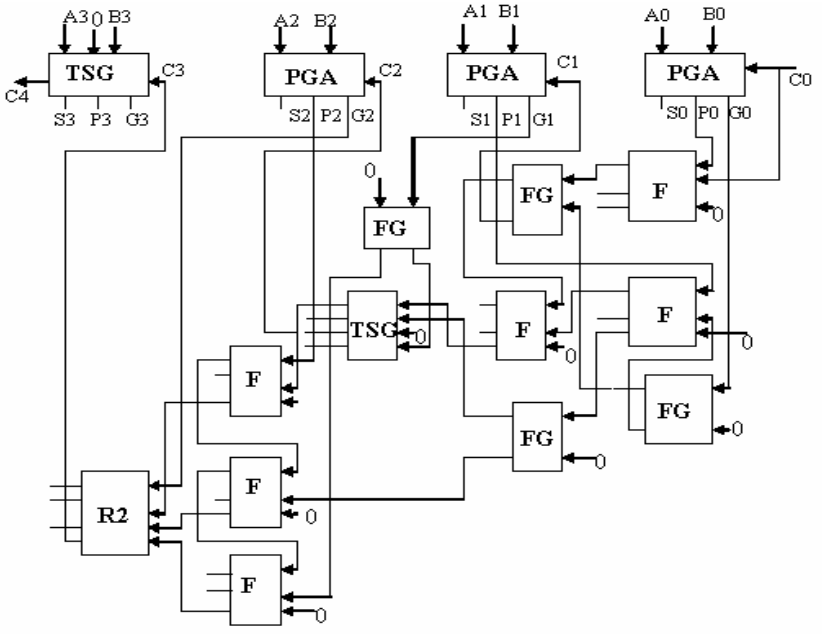


Fig. 14. Reversible CLA Designed with TSG, Fredkin(F), Feynman(FG) and R2 gate

Table 5. A Comparison of Results of Reversible CLA

	Number of Reversible Gates Used	Number of Garbage Outputs Produced
Proposed Circuit	19	0
Existing Circuit[16]	22	2

5 Conclusions

This paper presents a new reversible 4*4 gate called TSG gate that has been used to design optimized adder architectures like ripple carry adder, BCD adder and carry look ahead adder. It is proved that the adder architectures using the proposed TSG gate are better than the existing ones in literature in terms of reversible gates and garbage outputs. Reversible logic finds its application in areas such as quantum computing, nano-technology, and optical computing and the proposed TSG gate and efficient adder architectures that have been derived thereof are one of the contributions to reversible logic. The proposed circuits can be used for designing

reversible systems that are optimized in terms of area and power consumption. The reversible gate proposed and circuits designed using this gate form the basis of an ALU of a primitive quantum CPU.

References

1. Landauer, R.: Irreversibility and Heat Generation in the Computational Process. IBM Journal of Research and Development, 5,(1961) 183-191
2. Bennett, C.H.: Logical Reversibility of Computation. IBM J. Research and Development. (Nov 1973) 525-532
3. Fredkin, E., Toffoli, T.: Conservative Logic. International Journal of Theor. Physics. (1982) 219-253
4. Toffoli,T. :Reversible Computing. Tech memo MIT/LCS/TM-151. MIT Lab for Computer Science (1980).
5. LEPORATI, Alberto., ZANDRON, Claudio., MAURI, Giancarlo.: Simulating the Fredkin Gate with Energy Based P Systems. Journal of Universal Computer Science,Vol. 10, Issue 5. 600-619
6. Khan, Md. M. H Azad.: Design of Full-adder With Reversible Gates. International Conference on Computer and Information Technology, Dhaka, Bangladesh. (2002) 515-519
7. 7. Babu, Hasan, Hafiz Md., Islam, Md. Rafiqul., Chowdhury, Ali, Mostahed, Syed., and Chowdhury, Raja, Ahsan.: Reversible Logic Synthesis for Minimization of Full Adder Circuit. Proceedings of the EuroMicro Symposium on Digital System Design (DSD'03), Belek-Antalya, Turkey. (Sep 2003) 50-54
8. 8. Babu, Hasan, Hafiz Md., Islam, Md. Rafiqul., Chowdhury, Ali, Mostahed, Syed., and Chowdhury, Raja, Ahsan.: Synthesis of Full-Adder Circuit Using Reversible Logic. Proceedings 17th International Conference on VLSI Design (VLSI Design 2004), Mumbai, India. (Jan 2004) 757-760
9. Bruce, J.W.,Thornton, M.A., Shivakumariah, L., Kokate, P.S. and Li, X.: Efficient Adder Circuits Based on a Conservative Logic Gate. Proceedings of the IEEE Computer Society Annual Symposium on VLSI(ISVLSI'02), Pittsburgh, PA, USA. (April 2002) 83-88
10. Schrom, G.: Ultra Low Power CMOS Technology. PhD Thesis, Technischen Universitat Wien.(June 1998)
11. Knil, E., Laflamme, R. and Milburn, G.J.: A Scheme for Efficient Quantum Computation with Linear Optics. Nature. (Jan 2001) 46-52
12. M. Nielsen and I. Chaung, "Quantum Computation and Quantum Information", Cambridge University Press, 2000.
13. Merkle, R.C.: Two Types of Mechanical Reversible Logic. Nanotechnology 4. (1993) 114-131
14. Vedral, Vlatko.,Bareno,Adriano. and Artur Ekert.: Quantum Networks for Elementary Arithmetic Operations. arXiv:quant-ph/9511018 v1. (nov 1995)
15. Babu, Hasan, Hafiz Md. and Chowdhury, Raja, Ahsan.: Design of a Reversible Binary Coded Decimal Adder by Using Reversible 4-Bit Parallel Adder. 18th International Conference on VLSI Design (VLSI Design 2005), Kolkata, India.(Jan 2005) 255-260
16. Chung,Wen,Kai. and Tseng,Chien-Cheng.: Quantum Plain and Carry Look Ahead Adders. arxiv.org/abs/quant-ph/0206028.
17. 17. Thapliyal, Himanshu., Srinivas, M.B. and Arabnia,R.,Hamid.: A Need of Quantum Computing, Reversible Logic Synthesis of Parallel Binary Adder-Subtractor. The 2005 International Conference on Embedded System and Applications(ESA'05), Las Vegas, U.S.A. (June 2005) 60-67

18. Thapliyal, Himanshu., Srinivas, M.B. and Arabnia, R., Hamid.: A Reversible Version of 4 x 4 Bit Array Multiplier With Minimum Gates and Garbage Outputs. The 2005 International Conference on Embedded System and Applications(ESA'05), Las Vegas, U.S.A. (June 2005) 106-114
19. Thapliyal, Himanshu., Srinivas, M.B. and Arabnia ,R., Hamid.: Reversible Logic Synthesis of Half, Full and Parallel Subtractors.The 2005 International Conference on Embedded System and Applications (ESA'05), Las Vegas, U.S.A. (June 2005) 165-172
20. Vasudevan,D.P., Lala P.K. and Parkerson, J.P.: Online Testable Reversible Logic Circuit Design using NAND Blocks. Proc, Symposium on Defect and Fault Tolerance. (Oct 2004) 324-331

Implementation and Analysis of TCP/IP Offload Engine and RDMA Transfer Mechanisms on an Embedded System*

In-Su Yoon and Sang-Hwa Chung

Department of Computer Engineering,
Pusan National University
{isyoon, shchung}@pusan.ac.kr

Abstract. The speed of present-day network technology exceeds a gigabit and is developing rapidly. When using TCP/IP in these high-speed networks, a high load is incurred in processing TCP/IP protocol in a host CPU. To solve this problem, research has been carried out into TCP/IP Offload Engine (TOE) and Remote Direct Memory Access (RDMA). The TOE processes TCP/IP on a network adapter instead of using a host CPU; this reduces the processing burden on the host CPU, and RDMA eliminates any copy overhead of incoming data streams by allowing incoming data packets to be placed directly into the correct destination memory location. We have implemented the TOE and RDMA transfer mechanisms on an embedded system. The experimental results show that TOE and RDMA on an embedded system have considerable latencies despite of their advantages in reducing CPU utilization and data copy on the receiver side. An analysis of the experimental results and a method to overcome the high latencies of TOE and RDMA transfer mechanisms are presented.

1 Introduction

Ethernet technology is widely used in many areas. The Internet already uses bandwidths of one gigabit per second, and recently a 10 gigabit Ethernet bandwidth was standardized. TCP/IP is the most popular communication protocol for the Ethernet and is processed on a host CPU in all computer systems. This imposes enormous loads on the host CPU [1]. Moreover, the load on the host CPU increases as the physical bandwidth of a network is improved. To solve this problem, much research has been carried out using TCP/IP Offload Engine (TOE), in which TCP/IP is processed on a network adapter instead of a host CPU. TOE can reduce the high load imposed on a host CPU and can help the CPU concentrate on executing its jobs (except communication).

* This work was supported by the Regional Research Centers Program(Research Center for Logistics Information Technology), granted by the Korean Ministry of Education & Human Resources Development.

T. Srikanthan et al. (Eds.): ACSAC 2005, LNCS 3740, pp. 818–830, 2005.

Although TOE reduces much of the TCP/IP protocol processing burden on the main CPU, the ability to perform zero copy of incoming data streams on a TOE is dependent on the design of the TOE, the operating systems programming interface, and the applications communication model. In many cases, a TOE does not directly support zero copy of incoming data streams. To solve this problem, a Remote Direct Memory Access (RDMA) consortium [2] proposed the use of the RDMA mechanism. RDMA allows one computer to directly place data into another computers memory with minimum demands on the memory bus bandwidth and CPU processing overhead. In the RDMA mechanism, each incoming network packet has enough information to allow its data payload to be placed directly into the correct destination memory location, even when the packets arrive out of order. The direct data placement property of RDMA eliminates intermediate memory buffering and copying, and the associated demands on the memory and processor resources of the computing nodes, without requiring the addition of expensive buffer memory on the Ethernet adapter. In addition the RDMA uses the existing IP/Ethernet based network infrastructure. An RDMA-enabled Network Interface Controller (RNIC) can provide support for the RDMA over a TCP protocol suite and include a combination of TOE and RDMA functions in the same network adapter.

We implemented a TOE on an embedded system using Linux. As preliminary research into implementing an RNIC, we also implemented an RDMA transfer mechanism on the embedded system. Although the RDMA transfer mechanism was not combined with the TOE in the current implementation, it worked as a transport layer by itself, and supported TCP/IP applications without code modification. We performed experiments using the TOE and RDMA transfer mechanisms on the embedded system. Results of these experiments exhibited high latencies despite having the advantages of reducing CPU utilization and data copy on the receiver side. We have analyzed the results and in this paper propose a method to overcome the high latencies of TOE and RDMA.

2 Related Work

Research has been conducted into analyzing TCP/IP overheads [3, 4] and solving this problem [5, 6, 7]. Recent research into the TOE technology has been active. Also there are RDMA implementations, including RDMA over InfiniBand [8] and RDMA over VIA [9] architectures, but RNIC, which is a combination form of TOE and RDMA, is not available yet.

There have been two approaches used to develop TOE. In the first approach, an embedded processor on a network adapter processes TCP/IP using software. Intel's PRO1000T IP Storage Adapter [10] is example of research into software-based TOE. An advantage of this approach is its easy implementation but it has a disadvantage in network performance. According to experiments performed at Colorado University [11], the unidirectional bandwidth of this adapter does not exceed 30 MB/s, which is about half the bandwidth of 70 MB/s achieved by general gigabit Ethernet adapters. The second approach is to develop a special-

ized ASIC to process TCP/IP. Alacritech's session-layer interface control (SLIC) architecture [12] and Adaptec's network accelerator card (NAC) 7211 [13] are examples of research into hardware-based TOE. The bandwidth of Alacritech's SLIC is over 800 Mbps. This shows that the hardware-based approach guarantees network performance, but this approach has shortcomings in flexibility. Therefore, it is hard to add new application layers on top of TOE to make their network adapters stand alone for applications such as iSCSI. Moreover, this approach has a difficulty in implementing complicated session layers and IPSEC algorithms, which are mandatory in IPv6, by hardware logic.

To overcome these flexibility problems, we used an embedded processor to implement TOE and RDMA. However, this approach has a disadvantage in network performance as explained before. To improve the performance of the software-based approach, we analyzed experimental results and proposed two different implementation methods of TOE and RDMA.

3 Implementation of TOE and RDMA Transfer Mechanisms

We implemented the TOE and RDMA mechanisms to support TCP/IP applications using a socket library without any code modification. For this purpose, we modified the Linux kernel of the host so that the kernel did not process the TCP/IP protocol stack. We also implemented the TOE and RDMA transfer mechanisms on a network card. The implementations and mechanisms of the host side and the card side are presented in Section 3.1 and 3.2, respectively.

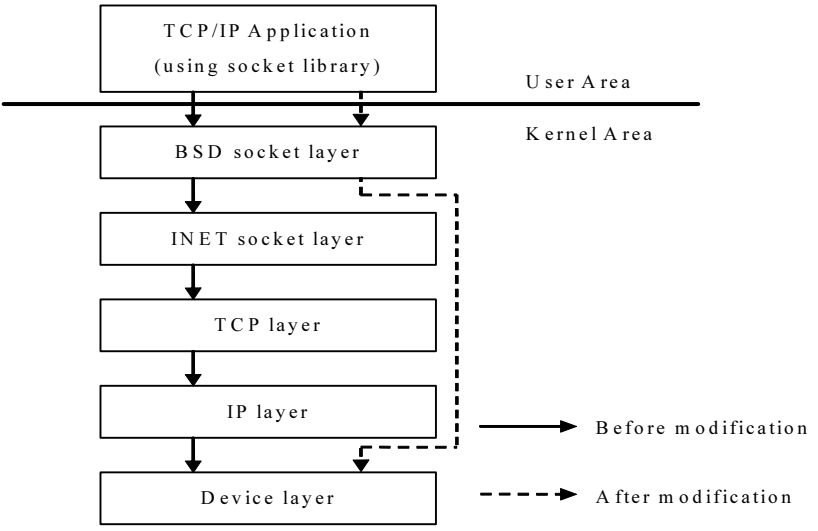


Fig. 1. A comparison of the original TCP/IP protocol stack and the modified stack

3.1 Implementation of the Host Side

In this section, we will explain how we can use TOE without carrying out any modification to existing TCP/IP application programs. For this purpose, we modified the Linux kernel 2.4.27. The modified Linux kernel sources were the netdevice.h and socket.h files in the include/linux directory and the socket.c file in the net/ directory. Figure 1 shows a comparison of the TCP/IP protocol hierarchy before and after modification.

TCP/IP applications use a normal socket library. The BSD socket layer passes the socket operations directly to the device layer as shown in Figure 1. Then, the device layer processes the operations. TCP/IP applications do not need to be rewritten to take advantage of the TOE. The only thing required is that users define the TOE in the socket creation code.

3.2 Implementation of the Card Side

We used a Cyclone Microsystems PCI-730 Gigabit Ethernet controller [14] to implement the TOE and RDMA transfer mechanisms. The PCI-730 has an Intel 600 MHz XScale CPU and gigabit interface that is based on Intel's 82544 chip. Figure 2 shows a block diagram of the PCI-730 card.

We also used a Linux kernel 2.4.18 for the embedded operating system, a GNU XScale compiler for implementing the embedded kernel module and embedded application. The TOE and RDMA transfer mechanisms are discussed in following paragraphs.

TOE Transfer Mechanism. The embedded application on the embedded Linux processes the TCP/IP and the embedded kernel module controls the

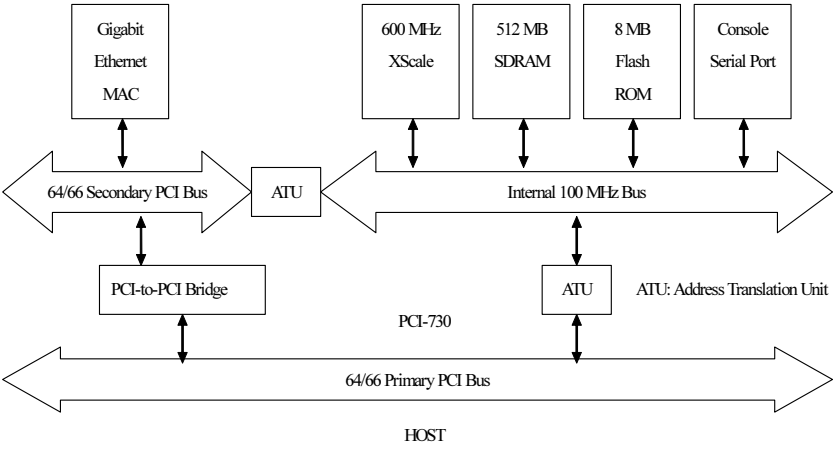


Fig. 2. A block diagram of the PCI-730 card

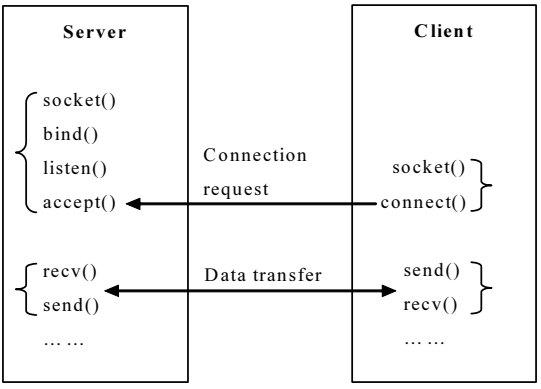


Fig. 3. A typical server and client model

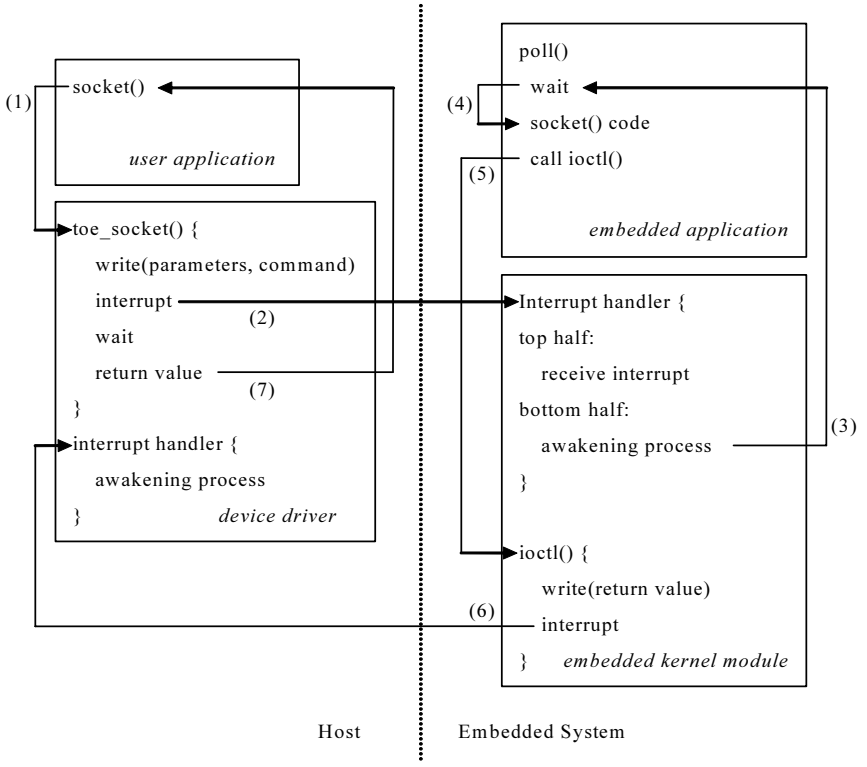


Fig. 4. TOE processing mechanism related to TCP/IP connection

interface between the host and the embedded application. Figure 3 shows a typical TCP/IP server and client model, and Figures 4 and 5 represent how the TOE processes the typical server/client model.

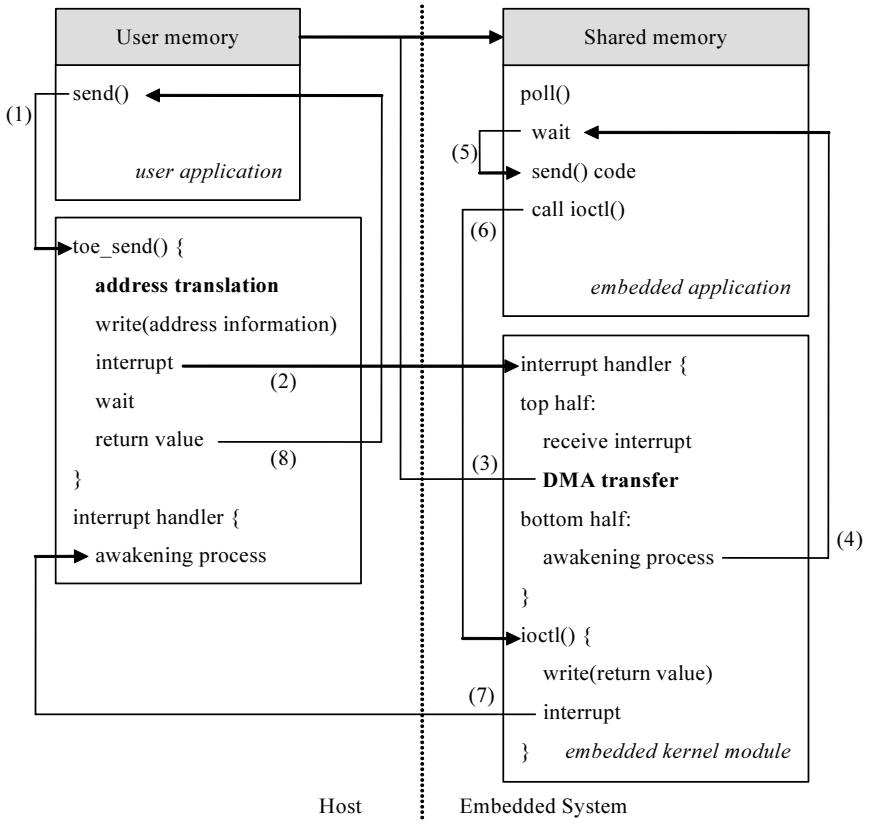


Fig. 5. TOE processing mechanism related to data transfer

As shown in Figure 3, functions that form the socket library can be divided into two groups: functions for the TCP/IP connection and functions for data transfer. Figure 4 shows how the functions for the TCP/IP connection are processed on the TOE.

Figure 4 shows how the socket() function, which is used in socket creation, is processed. Functions, such as bind() and listen() are processed in the same way. When a host side's user application calls the socket() function, the BSD socket layer calls the toe_socket() function, which is implemented in the device driver (1). The toe_socket() function passes user parameters and a command to the embedded kernel module. The command is used to identify which socket functions must be processed in the embedded application. Then, the device driver interrupts kernel module (2) and the toe_socket() function pauses. After receiving the interrupt, the embedded kernel module activates embedded application (3). The embedded application executes the relevant codes according to the given command (4). After executing the socket() code, the application delivers the return value of the socket() to the embedded kernel module through the ioctl()

function (5). The embedded kernel module writes the return value to the device driver and interrupts the device driver (6). The interrupt reactivates the toe_socket() function. Finally, the toe_socket() function returns the value to the host's user application (7).

Data transfer functions, such as send() and recv(), add two additional jobs to the processing mechanism described above. One is to deliver the address information of the user memory to the embedded kernel module. The other is the DMA transfer using that information. Figure 5 shows the TOE processing mechanism related to data transfer.

Figure 5 shows how the send() function is processed. When the send() function is called by the user application (1), the device driver does not copy the data from the user memory into the device driver. The device driver only calculates the physical address and length of the user memory. After that, the device driver passes the address and length pairs to the embedded kernel module and interrupts the embedded kernel module (2). The embedded kernel module loads the address information onto the DMA controller. The DMA controller transfers the data of the user memory to the memory on the PCI-730 card shared by the embedded kernel module and the embedded application (3). Then, the embedded kernel module activates the embedded application (4). The embedded application sends out the data in the shared memory using the send() function (5). After sending out the data, the mechanism of the returning value is identical to that shown in Figure 4.

In this implementation, the copy from the user memory to the device driver is not needed because the device driver only calculates the physical address and length of the user memory. By just passing the address information and sharing memory between the embedded kernel module and the embedded application, three copies are reduced to only one DMA transfer. These three copies are the copy from the user memory to the device driver, the copy from the device driver to the embedded kernel module and the copy from the embedded kernel module to the embedded application. The recv() function is also processed similarly.

In this paper, we used the embedded application to execute the socket functions according to the given command by the device driver. These socket functions are actually processed in the Linux kernel when the embedded application invokes system calls such as socket(), send() and recv(). Although the Linux kernel processes complicated low-level functions such as managing established socket connections, there exist many other functions including bind() and listen() to complete the whole TOE process. We have implemented such functions in the embedded application level.

RDMA Transfer Mechanism. As preliminary research into implementing an RNIC, an RDMA transfer mechanism was also implemented on the embedded system. Although the RDMA transfer mechanism was not combined with the TOE in the current implementation, it worked as a transport layer by itself, and supported TCP/IP applications without code modification. The mechanism that is used to process the TCP/IP connections is identical to that shown in Figure 4. Data transfer functions, such as the recv() and send() functions, are

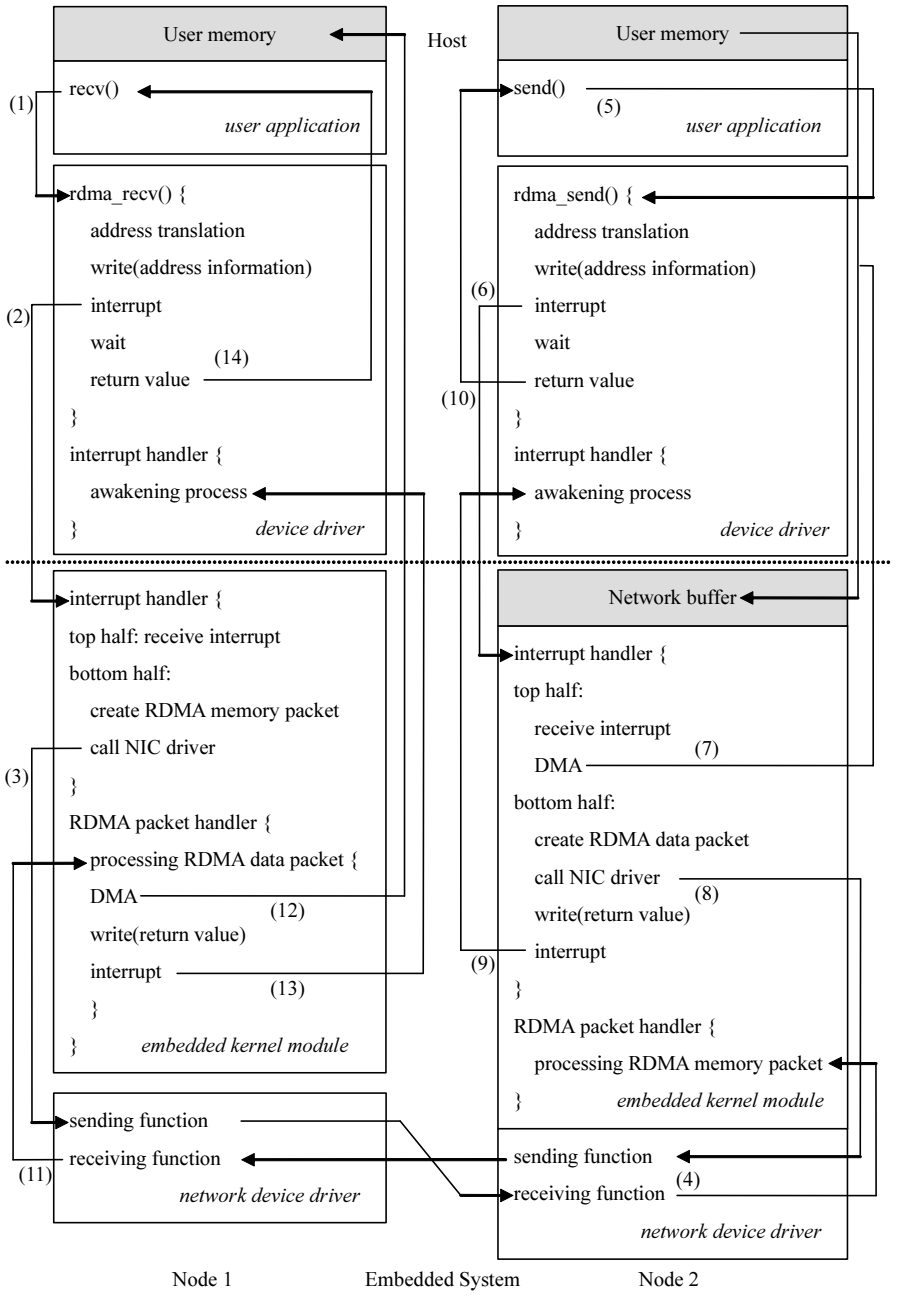


Fig. 6. Processing mechanism for the data transfer through the RDMA write

processed through the RDMA write. Figure 6 shows the processing mechanism for the data transfer through the RDMA write.

When data is transferred using RDMA as shown in Figure 6, the existing TCP/IP is not used in an embedded system. Instead of using TCP/IP, RDMA works as a transport layer by itself. The transport layer makes an outgoing RDMA packet and processes an incoming RDMA packet using the RDMA packet handler which is called when the RDMA packet arrives. Since the RDMA packet processing mechanism was implemented in the embedded kernel module, the embedded application was not involved in data transfer. As shown in Figure 5, the TOE transfer mechanism uses the embedded application for the data transfer. Because of this approach, the TOE transfer mechanism unavoidably has a context switching overhead between the embedded kernel module and the embedded application. On the other hand, the RDMA transfer mechanism does not have the context switching overhead. In the following paragraphs, we present the RDMA transfer mechanism in detail.

The processing sequence from calling the recv() function in the user application (1) to interrupting the embedded kernel module (2) is identical to the TOE transfer mechanism. After that, the interrupt handler in the embedded kernel module creates a socket buffer and fills a header of the socket buffer with the MAC addresses of the source and destination. The data payload of the packet is filled with the physical address and length pairs of user memory. We denote this packet type as an "RDMA memory packet". The embedded kernel module passes such a packet to the network device driver, and the network device driver sends the RDMA memory packet to Node 2 (3).

When the RDMA packet handler of Node 2 receives the RDMA memory packet from Node 1, Node 2's embedded kernel module stores the packet and manages the remote memory information of Node 1 (4). The processing sequence from calling the send() function in the user application (5) to interrupting the embedded kernel module (6) is also identical to the TOE transfer mechanism. After that, the embedded kernel module loads the physical address and length pairs of the user memory onto the DMA controller. The DMA controller transfers the data of the user memory to the network buffer in the embedded kernel module (7). The processing sequence from creating a socket buffer to making a packet header is identical to that described above. The data payload of the packet is filled with the remote memory information of Node 1, which is managed in the embedded kernel module, and then, the data of the network buffer is added to the packet. We denote this packet type as an "RDMA data packet". Then, the RDMA data packet is transferred to Node 1 (8).

When the RDMA packet handler of Node 1 receives the RDMA data packet from Node 2 (11), the handler retrieves the memory information from the packet. The DMA controller transfers the actual data in the packet to the user memory using that information (12). Because the RDMA data packets have the memory information for their destination, the data in the packets can be transferred correctly to the user memory, although packets may be dropped or be arranged out of order due to network congestion.

4 Experimental Results and Analysis

We measured the elapsed times for each operation of the TOE and RDMA transfer mechanisms using the embedded system. In this section, we analyze the experimental results and explain how much overhead each operation has. Based on our analysis of the experimental results, we propose efficient TOE and RNIC models, which are able to overcome the overhead.

For our experiments, we used two Pentium IV servers equipped with a PCI-730 card and connected two computers using 3COM's SuperStack3 switch. We measured the latency from calling the send() function until four bytes of data were placed into the receiver's memory. In addition, we divided the latencies into the elapsed times for each operation of the TOE and RDMA transfer mechanisms. The experimental results of the TOE are shown in Table 1.

As shown in Table 1, the latency of the TOE was 533 μs. This is a very high latency and is not adequate to support gigabit Ethernet speeds. The operations shown in Table 1 can be divided into three groups. The first group is the interrupt and DMA operations. Reducing the elapsed times for these operations is difficult because these are related to hardware concerns. The second group is the TCP/IP and packet processing operations. These operations are essentially needed to process the TCP/IP in the TOE. The third group is the operations related to the interface between the embedded kernel module and the embedded application.

The operations of the third group can be eliminated if the data transfer functions such as send() and recv() can be processed in the embedded kernel module instead of the embedded application. To do so, the embedded application level functions such as socket(), send() and recv() must be implemented inside the Linux kernel as a module, and the whole TOE process must be completed using only the Linux kernel functions.

If the embedded kernel module processes the send() and recv() functions without the embedded application, the operations that are indicated in the bold-faced

Table 1. The elapsed time for each operation of the TOE (units = μs)

Operation	Time	%
Address translation in the device driver	4	1
Interrupt processing (Host↔Card)	10	2
Data and memory information transfer by DMA (send)	37	7
Waiting for scheduling the interrupt handler's bottom half	31	6
Awakening the embedded application	**31**	**6**
Processing TCP/IP protocol in the embedded kernel module	249	47
Packet processing in the network device driver	84	16
Entering ioctl() in the embedded kernel module after executing send() in the embedded application	**27**	**5**
Entering ioctl() in the embedded kernel module after executing recv() in the embedded application	**27**	**5**
Data transfer by DMA (receive)	33	6
Total	533	100

Table 2. The elapsed time for each operation of the RDMA (units = μs)

Operation	Time	%
Address translation in the device driver	4	1
Interrupt processing (Host↔Card)	10	4
Data and memory information transfer by DMA (send)	37	14
Waiting for scheduling the interrupt handler's bottom half	31	12
Processing RDMA packet in the embedded kernel module	68	25
Packet processing in the network device driver	84	31
Data transfer by DMA (receive)	33	12
Total	267	100

text in Table 1 are not needed. In addition, this approach also eliminates the copy overhead from the embedded application to the embedded kernel module. According to Ref. [15], the copy overhead from user to kernel space occupies 17% of the time required to process the TCP/IP. Therefore, if the embedded kernel module processes the send() and recv() functions instead of the embedded application, then the TOE can have a latency of 406 μs.[1] This can reduce the time by 24% versus the implemented TOE in this research. The RDMA transfer mechanism processes the RDMA packet in the embedded kernel module without an embedded application. This mechanism is similar to the proposed method discussed in the previous paragraph. The difference is that the RDMA transfer mechanism does not use TCP/IP, but use its own packet processing routines to form the RDMA packet. The experimental results of the RDMA are shown in Table 2.

Because the RDMA transfer mechanism does not use the embedded application, the context switching overhead between the embedded kernel module and the embedded application does not exist. Also the RDMA transfer mechanism uses its own packet processing routines instead of the complicated TCP/IP. Because of these aspects, the RDMA transfer mechanism had a latency of 267 μs, as shown in Table 2. Based on these experimental results, we estimate that an RNIC that is a combination of the TOE and RDMA will have a latency of 474 μs. This is a sum of the 406 and 68 μs latencies. The latency of 406 μs derives from the TOE using the method discussed earlier to reduce the latency. The latency of 68 μs derives from the RDMA packet processing time shown in Table 2. Because other operations in Table 2 are processed by the TOE, The elapsed time of 68 μs is only added to the latency of the TOE.

There is another method that can reduce the latency. This is not to use an operating system such as Linux kernel for the embedded system. To apply this method to implement the TOE and RNIC, two things are required. First, a detached TCP/IP code from the operating system, such as a lightweight TCP/IP stack (lwIP) [16], is required. This can process the TCP/IP without an operating system. Second, a program to control the hardware without an operating system is also required. The example is Intel's PRO1000T IP Storage Adapter [10]. This uses its own program to control the hardware and process the TCP/IP without

[1] $406 = 533 - \{(31 + 27 + 27) + (249 * 0.17)\}$.

an operating system. If this method is applied to implement the TOE and RNIC, the copy overhead between the TCP/IP layer and the network device driver layer can be eliminated. In addition, any unnecessary complicated algorithms and data structures, which are used in the operating system for controlling the embedded systems, can be simplified or eliminated. According to Ref. [15], the copy overhead from the TCP/IP layer to the network device driver layer occupies 32% of the time required to process the TCP/IP and the operating system overhead occupies 20% of the time required to process the TCP/IP. Therefore, a TOE that does not have these overheads is able to have a latency of 276 μs.[2] In the same way, we can estimate the reduced latency of the RNIC. The copy overhead also occurs when the RDMA packet is passed to the network device driver. If we assume that the operating system overhead of processing the RDMA packet is same as that of processing the TCP/IP (20%), and an RNIC that does not have these overheads is able to have a latency of 309 μs.[3]

In the work described in this paper, we implemented TOE and RDMA transfer mechanisms using the PCI-730 card. The card has a 600 MHz XScale CPU and an internal 100 MHz bus. Because of the slower bus speed than the CPU speed, the CPU has to wait six clocks whenever cache misses occur. Because TCP/IP processing requires lots of memory references and copies, then this overhead is important in implementing the TOE and RNIC. Therefore, if we use an up-to-date chip [17] that has 800 MHz XScale CPU and internal 333 MHz bus, the TOE and RNIC models proposed in this paper will be able to have a latency of near 200 μs. Among the commercial TOE products, a hardware-based TOE product [18] has a latency of 60 μs and a bandwidth of 2.3 Gbps by fully offloading the TCP/IP processing to ASIC. Therefore, the proposed TOE and RNIC models that have a latency of around 200 μs will be able to have a higher bandwidth than 600 Mbps.

5 Conclusions and Future Work

We have implemented TOE and RDMA transfer mechanisms using the PCI-730 card and the embedded Linux. Also, we have explained their mechanisms in this paper. The TOE and RDMA transfer mechanisms had latencies of 533 and 267 μs, respectively. We analyzed the experimental results by dividing the latencies into the elapsed times for each operation of the TOE and RDMA transfer mechanisms. Based on this analysis, we proposed two different implementation methods to reduce the latencies of the TOE and RNIC. The first method was that the embedded kernel module processes the data transfer functions instead of the embedded application. The second method was to implement the TOE and RNIC without an operating system. For each proposed model, we estimated the time that could be reduced, and showed the expected latencies. In future work, we will implement the TOE and RNIC without using an operating system and we will use the latest high performance hardware instead of the PCI-730.

[2] $276 = 406 - 249 * (0.32 + 0.2)$.
[3] $309 = 474 - \{(249 + 68) * (0.32 + 0.2)\}$.

References

[1] N. Bierbaum, "MPI and Embedded TCP/IP Gigabit Ethernet Cluster Computing", Proc. of the 27th Annual IEEE Conference on Local Computer Networks 2002, pp. 733–734, Nov 2002.
[2] http://www.rdmaconsortium.org/home
[3] J. Kay, J. Pasquale, "Profiling and reducing processing overheads in TCP/IP", IEEE/ACM Transactions on Networking, Vol. 4, No. 6, pp. 817–828, Dec 1996.
[4] S-W. Tak, J-M. Son, and T-K. Kim, "Experience with TCP/IP networking protocol S/W over embedded OS for network appliance", Proc. of 1999 International Workshops on Parallel Processing, pp. 556–561, Sep 1999.
[5] P. Camarda, F. Pipio, and G. Piscitelli, "Performance evaluation of TCP/IP protocol implementations in end systems", IEEE Proc. of Computers and Digital Techniques, Vol. 146, No. 1, pp. 32–40, Jan 1999.
[6] H. T. Kung and S. Y. Wang, "TCP trunking: design, implementation and performance", Proc. of 7th International Conference on Network Protocols 1999, pp. 222–231, Oct 1999.
[7] H. Bilic, Y. Birk, I. Chirashnya, and Z. Machulsky, "Deferred segmentation for wire-speed transmission of large TCP frames over standard GbE networks", Hot Interconnects 9, pp. 81–85, Aug 2001.
[8] J. Liu, D. K. Panda, and M. Banikazemi, "Evaluating the Impact of RDMA on Storage I/O over InfiniBand", SAN-03 Workshop (in conjunction with HPCA), Feb 2004.
[9] H. Hellwagner and M. Ohlenroth, "VI architecture communication features and performance on the Gigabit cluster LAN", Future Generation Computer Systems, Vol. 18, Issue 3, Jan 2002.
[10] http://support.intel.com/support/network/adapter/iscsi/tip/index.htm
[11] S. Aiken, D. Grunwald, A. R. Pleszkun, and J. Willeke, "A Performance Analysis of the iSCSI Protocol", Proc. of the 20th IEEE Symposium on Mass Storage Systems 2003, April 2003.
[12] http://www.alacritech.com/html/tech_review.html
[13] http://www.adaptec.com/pdfs/NAC_techpaper.pdf
[14] http://www.cyclone.com/products/pci730.htm
[15] D. D. Clark, V. Jacobson, J. Romkey, and H. Salwen, "An Analysis of TCP Processing Overhead", IEEE Communications, Vol. 27, No. 6, pp. 23–29, Jun 1989.
[16] A. Dunkels, "Full TCP/IP for 8-Bit Architectures", In Proc. of the First International Conference on Mobile Applications, Systems and Services, San Francisco, CA, USA, May 2003.
[17] http://www.intel.com/design/iio/iop333.htm
[18] H. Ghadia, "Benefits of full TCP/IP offload in the NFS Services", 2003 NFS Industry Conference, Sep 2003.

Author Index