Pen-Chung Yew
Jingling Xue (Eds.)

LNCS 3189

Advances in Computer Systems Architecture

9th Asia-Pacific Conference, ACSAC 2004
Beijing, China, September 2004
Proceedings

Springer

Lecture Notes in Computer Science 3189

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

Pen-Chung Yew   Jingling Xue (Eds.)

Advances in Computer Systems Architecture

9th Asia-Pacific Conference, ACSAC 2004
Beijing, China, September 7-9, 2004
Proceedings

Springer

Volume Editors

Pen-Chung Yew
University of Minnesota at Twin Cities
Department of Computer Science and Engineering
Twin Cities, MN, USA
E-mail: yew@cs.umn.edu

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

Library of Congress Control Number: 2004111113

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

ISSN 0302-9743
ISBN 3-540-23003-3 Springer Berlin Heidelberg New York

Springer is a part of Springer Science+Business Media

springeronline.com

Typesetting: Camera-ready by author, data conversion by PTP-Berlin, Protago-TeX-Production GmbH
Printed on acid-free paper SPIN: 11318651 06/3142 5 4 3 2 1 0

Preface

On behalf of the program committee, we were pleased to present this year's program for *ACSAC: Asia-Pacific Computer Systems Architecture Conference.*

Now in its ninth year, ACSAC continues to provide an excellent forum for researchers, educators and practitioners to come to the Asia-Pacific region to exchange ideas on the latest developments in computer systems architecture. This year, the paper submission and review processes were semiautomated using the free version of CyberChair. We received 152 submissions, the largest number ever. Each paper was assigned at least three, mostly four, and in a few cases even five committee members for review. All of the papers were reviewed in a two-month period, during which the program chairs regularly monitored the progress of the review process. When reviewers claimed inadequate expertise, additional reviewers were solicited. In the end, we received a total of 594 reviews (3.9 per paper) from committee members as well as 248 coreviewers whose names are acknowledged in the proceedings. We would like to thank all of them for their time and effort in providing us with such timely and high-quality reviews, some of them on extremely short notice.

After all of the reviews were received, there was a one-week electronic program committee meeting during May 14 and May 21. All of the papers were reviewed and discussed by the program committee, and the final set of papers were selected. Program committee members were allowed to submit papers, but their papers were handled separately. Each of their papers was assigned to at least four committee members and reviewed under the same rigorous review process. The program committee accepted 7 out of 11 "PC" submissions. In the end, the program committee selected a total of 45 papers for this year's program with an acceptance rate close to 30%. Unfortunately, many fine papers could not be accommodated in this year's program because of our schedule.

In addition to the contributed papers, this year's program included invited presentations. We were very pleased that three distinguished experts accepted our invitation to share their views on various aspects of computer systems architecture design: James E. Smith (University of Wisconsin-Madison, USA) on *Some Real Observations on Virtual Machines*, Jesse Z. Fang (Intel, USA) on *A Generation Ahead of Microprocessor: Where Software Can Drive uArchitecture to?*, and, finally, Guojie Li (Chinese Academy of Sciences, China) on *Make Computers Cheaper and Simpler.*

On behalf of the program committee, we thank all of the authors for their submissions, and the authors of the accepted papers for their cooperation in getting their final versions ready in time for the conference. We would also like to thank the Web Chair, Lian Li, for installing and maintaining CyberChair, and the Local Arrangements Chair, Wenguang Chen, for publicizing this conference.

Finally, we want to acknowledge the outstanding work of this year's program committee. We would like to thank them for their dedication and effort in providing timely and thorough reviews for the largest number of submissions ever in our conference history, and their contribution during the paper selection process. It was a great pleasure working with these esteemed members of our community. Without their extraordinary effort and commitment, it would have been impossible to put such an excellent program together in a timely fashion. We also want to thank all our sponsors for their support of this event. Last, but not least, we would like to thank the General Chair, Weimin Zheng for his advice and support to the program committee and his administrative support for all of the local arrangements.

June 2004

Pen-Chung Yew
Jingling Xue

Conference Organization

General Chair

Weimin Zheng	Tsinghua University, China

Program Chairs

Pen-Chung Yew	University of Minnesota, USA
Jingling Xue	University of New South Wales, Australia

Local Arrangements Chair

Wenguang Chen	Tsinghua University, China

Local Arrangements Committee

Hongliang Yu	Tsinghua University, China
Jianian Yan	Tsinghua University, China
Jidong Zhai	Tsinghua University, China
Ruini Xue	Tsinghua University, China
Jiao Lin	Tsinghua University, China

Web Chair

Lian Li	University of New South Wales, Australia

Program Committee

Lars Bengtsson	Chalmers University of Technology, Sweden
Sangyeun Cho	Samsung Electronics, Co., Korea
Lynn Choi	Korea University, Korea
Rudolf Eigenmann	Purdue University, USA
Jean-Luc Gaudiot	University of California, Irvine, USA
Antonio Gonzalez	Universitat Politecnica de Catalunya & Intel Labs, Spain
Gernot Heiser	National ICT Australia, Australia
Wei-Chung Hsu	University of Minnesota, USA
Chris Jesshope	University of Hull, UK
Angkul Kongmunvattana	University of Nevada, Reno, USA
Feipei Lai	National Taiwan University
Zhiyong Liu	National Natural Science Foundation of China, China
Guei-Yuan Lueh	Intel, USA
John Morris	Chung-Ang University, Korea & University of Auckland, New Zealand
Tadao Nakamura	Tohoku University, Japan
Yukihiro Nakamura	Kyoto University, Japan
Amos Omondi	Flinders University, Australia
Lalit M. Patnaik	Indian Institute of Science, Bangalore, India
Jih-Kwon Peir	University of Florida, USA
Ronald Pose	Monash University, Australia
Depei Qian	Xian Jiaotong University, China
Stanislav G. Sedukhin	University of Aizu, Japan
Naofumi Takagi	Nagoya University, Japan
Zhimin Tang	Chinese Academy of Sciences, China
Rajeev Thakur	Argonne National Laboratory, USA
Theo Ungerer	University of Augsburg, Germany
Winfried W. Wilcke	IBM Research, USA
Weng Fai Wong	National University of Singapore, Singapore
Chengyong Wu	Chinese Academy of Sciences, China
Ming Xu	National University of Defense Technology, China
Yuanyuan Yang	State University of New York at Stony Brook, USA
Rumi Zahir	Intel, USA
Chuanqi Zhu	Fudan University, China

Co-reviewers

Tom Adelmeyer
Alex Aletà
Jalal Almhana
Madhusudhanan Anantha
Juan Luis Aragon
Brian Armstrong
Eduard Ayguade
Faruk Bagci
Nitin Bahadur
Vishwanath P. Baligar
Bin Bao
Ayon Basumallik
Jürgen Beckeer
Ramón Beivide
Bryan Black
Tatiana Bokareva
Uwe Brinkschulte
Ralph Butler
Luis M. Díaz de Cerio
Jason Chang
Yen-Jen Chang
Mei Chao
Cheng Chen
Dong-Yuan Chen
Gen-Huey Chen
Haibo Chen
Howard Chen
Ronghua Chen
Tien-Fu Chen
Wen-Hsien Chen
Wenguang Chen
Yinwen Chen
Yung-Chiao Chen
Avery Ching
Seng-Cho Chou
Yang Wai Chow
Peter Chubb
C.G. Chung
Chung-Ping Chung
Sung Woo Chung
Josep M. Codina
Tim Conrad
Nawal Copty
Julita Corbalan
Toni Cortes
Alfredo Cristobal-Salas
Abhinav Das
Xiaotie Deng
Qiang Ding
Yingfei Dong
Klaus Dorfmüller-Ulhaas
David Du
Colin Egan
Kevin Elphinstone
Dongrui Fan
Hao Feng
Konrad Froitzheim
Rao Fu
Antonia Gallardo
Boon-Ping Gan
Enric Gibert
Marc Gonzalez
Charles Gray
Yi Guo
Weili Han
Wessam Hassanein
Guojin He
Gerolf Hoflehner
Scott Hoyte
Pao-Ann Hsiung
Wen Hu
Dandan Huan
Ing-Jer Huang
Junwei Huang
Lei Huang
Yi-Ping Hung
Wei Huo
Tomonori Izumi
Muhammad Mahmudul Islam
Yaocang Jia
Hong Jiang
Weihua Jiang
Yi Jiang
Troy A. Johnson
Edward Sim Joon
Sourabh Joshi
Roy Ju
Marcelo E. Kaihara

Dongsoo Kang
Ryosuke Kato
Jörg Keller
Ihn Kim
JinPyo Kim
Sunil Kim
Chung-Ta King
Jon Krueger
Fumio Kumazawa
Ihor Kuz
Atul Kwatra
Hsiu-Hui Lee
Hung-Chang Lee
Sanghoon Lee
Yong-fong Lee
Jianping Li
Jie Li
Shengjun Li
Wei Li
Yingsheng Li
Yunchun Li
Weifa Liang
Shih-wei Liao
Wanjiun Liao
Björn Liljeqvist
Ching Lin
Fang-Chang Lin
Fang-Pang Lin
Hung-Yau Lin
Shian-Hua Lin
Xiaobin Lin
Bin Liu
Chen Liu
Jiangchuan Liu
Jyi-shane Liu
Michael Liu
Tao Liu
Zhanglin Liu
Jiwei Lu
Peng Lu
Zhongzhi Luan
Jesus Luna
Yuh-Dauh Lyuu
Takahiko Masuzaki
Ryusuke Miyamoto

Chi Ma
Xiaosong Ma
Erik Maehle
Mike Mesnier
Neill Miller
Do Quang Minh
Dave Minturn
Steven Molnar
Rafael Moreno-Vozmediano
Alberto J. Munoz
Hashem Hashemi Najaf-abadi
Gil Neiger
Anindya Neogi
Tin-Fook Ngai
Qizhi Ni
Rong Ni
Hiroyuki Ochi
Robert Olson
Ming Ouhyoung
Deng Pan
Zhelong Pan
Marina Papatriantafilou
Chan-Ik Park
Gi-Ho Park
Junho Park
Enric Pastor
Jan Petzold
Matthias Pfeffer
Andy D. Pimentel
Dhiraj Pradhan
Nol Premasathian
Rolf Rabenseifner
Ryan Rakvic
Rajiv Ranjan
Xiaojuan (Joanne) Ren
Won Woo Ro
Shanq-Jang Ruan
Hariharan Sandanagobalane
Kentaro Sano
Hartmut Schmeck
Ioannis T. Schoinas
Peter Schulthess
André Seznec
Hemal Shah
Shrikant Shah

Hong Shen
Sameer Shende
Jang-Ping Sheu
Xudong Shi
Mon-Chau Shie
Yong Shin
Tsan-sheng Shsu
Siew Sim
Mostafa I. Soliman
James Stichnoth
Feiqi Su
Yeali Sun
Kosuke Tsujino
Hiroshi Tsutsui
Kazuyoshi Takagi
Akihito Takahashi
Shigeyuki Takano
Santipong Tanchatchawal
Wei Tang
Hariharan L. Thantry
Ekkasit Tiamkaew
Apichat Treerojporn
Kun-Lin Tsai
Sascha Uhrig
Gladys Utrera
Alexander Vazhenin
Xavier Vera
Murali Vilayannur
Harsh Vipat
Jian Wang
Kuochen Wang
Peng Wang
Qin Wang
ShengYue Wang
Xiaodong Wang
Ye Wang
Yanzhi Wen
Sung-Shun Weng
Adam Wiggins
Weng-Fai Wong
Hsiaokuang Wu
Hui Wu
Jiajun Wu
Jiesheng Wu
Meng-Shiou Wu
Youfeng Wu
CanWen Xiao
Dai Xiao
Junhua Xiao
Yang Xiao
Wenjun Xiao
Jinhui Xu
Chu-Sing Yang
Xu Yang
Zhen Yang
Handong Ye
Chingwei Yeh
Kyueun Yi
Heng Zhang
Hongjiang Zhang
Hui Zhang
Minjie Zhang
Weihua Zhang
Xiaomin Zhang
Xingjun Zhang
Zhenghao Zhang
Qin Zhao
Yili Zheng
Yuezhi Zhou
Jiahua Zhu

Table of Contents

Keynote Address I

Session 1A: Cache and Memory

Session 1B: Reconfigurable and Embedded Architectures

Session 2A: Processor Architecture and Design I

Session 2B: Power and Energy Management

Session 3A: Processor Architecture and Design II

Session 3B: Compiler and Operating System Issues

Keynote Address II

Session 4A: Application-Specific Systems

Session 4B: Interconnection Networks

Keynote Address III

Session 5A: Prediction Techniques

Session 5B: Parallel Architecture and Programming

Session 6A: Microarchitecture Design and Evaluations

Session 6B: Memory and I/O Systems

Session 7A: Potpourri

Some Real Observations on Virtual Machines

James E. Smith

Department of Electrical and Computing Engineering
University of Wisconsin-Madison
jes@ece.wisc.edu

Abstract. Virtual machines can enhance computer systems in a number of ways, including improved security, flexibility, fault tolerance, power efficiency, and performance. Virtualization can be done at the system level and the process level. Virtual machines can support high level languages as in Java, or can be implemented using a low level co-designed paradigm as in the Transmeta Crusoe. This talk will survey the spectrum of virtual machines and discuss important design problems and research issues. Special attention will be given to co-designed VMs and their application to performance- and power-efficient microprocessor design.

Replica Victim Caching to Improve Reliability of In-Cache Replication

W. Zhang

Dept of ECE, SIUC, Carbondale, IL 62901, USA
zhang@engr.siu.edu

Abstract. Soft error conscious cache design is a necessity for reliable computing. ECC or parity-based integrity checking techniques in use today either compromise performance for reliability or vice versa. The recently-proposed ICR (In-Cache Replication) scheme can enhance data reliability with minimal impact on performance, however, it can only exploit a limited space for replication and thus cannot solve the conflicts between the replicas and the primary data without compromising either performance or reliability. This paper proposes to add a small cache, called replica victim cache, to solve this dilemma effectively. Our experimental results show that a replica victim cache of 4 entries can increase reliability of L1 data caches 21.7% more than ICR without impacting performance, and the area overhead is within 10%.

1 Introduction and Motivation

Soft errors are unintended transitions of logic states caused by external radiations such as alpha particle and cosmic ray strikes. Recent studies [4,6,5,9] indicate that soft errors are responsible for a large percentage of computation failures. In current microprocessors, over 60% of the on-chip estate is taken by caches, making them more susceptible to external radiations. The soft errors in cache memories can easily propagate into the processor registers and other memory elements, resulting in catastrophic consequences on the execution. Therefore, soft error tolerant cache design is becoming increasingly important for failure-free computation.

Information redundancy is the main technique to improve data integrity. Currently the popular information redundancy scheme for memories is either byte-parity (one bit parity per 8-bit data) [1], or single error correcting-double error detecting (SEC-DED) code (ECC)[2,3]. However, both of these two schemes have deficiencies. Briefly, parity check can only detect single-bit errors. While SEC-DEC based schemes can correct single-bit errors and detect two-bit errors, they can also increase the access latency of the L1 cache, and thus not suitable for high-end processors clocked over 1GHz [7]. Recently, an approach called ICR (In-Cache Replication) has been proposed to enhance reliability of the L1 data cache for high-performance processors [7]. The idea of ICR is to exploit "dead" blocks in the L1 data cache to store the replicas for "hot" blocks so that a large percentage of read hits in the L1 can find their replicas in the same cache, which

P.-C. Yew and J. Xue (Eds.): ACSAC 2004, LNCS 3189, pp. 2–15, 2004.

can be used to detect and correct single bit and/or multiple bit errors. While the ICR approach can achieve a better tradeoff between performance and reliability than parity-only or ECC-only protection, it can only exploit a limited space (namely the "dead" blocks in the data cache) for replication. In addition, since the replica and the primary data are stored in the same cache, they inevitably have conflicts with each other. The current policy adopted by ICR [7] is to give priority to the primary data for minimizing the impact on performance. In other words, the data reliability is compromised. As illustrated in [7], 10% to 35% of data in the L1 data cache is not protected (i.e. having no replicas) by ICR schemes, which may cause severe consequences in computation and thus are not useful for applications that require high reliability or operate under highly noisy environments.

This paper proposes a novel scheme to enhance the reliability of ICR further by adding a small fully-associative cache to store the replica victims, which is called the replica victim cache in this paper. Unlike the victim cache proposed by jouppi [10] for reducing the conflict misses for a direct-mapped cache without In-Cache Replication, the proposed replica victim cache is utilized to store the replica victims, which are conflicting with the primary data or other replicas in the primary data cache, for enhancing reliability of the ICR approaches significantly without compromising performance. Moreover, since the replica is used to improve data integrity, the replica victim cache does not need to swap the replica with the primary data when accessed. In contrast, the traditional victim cache stores different data (i.e., victim) from the primary cache, and the victims need to be swapped to the L1 cache in the case of a miss in the L1 data cache that hits in the victim cache[10]. The paper examines the following problems.

1. How does reliability, in terms of loads with replica (see the definition in 4), relate to the size of the replica victim cache, the size and associativity of the primary cache? How much loads with replica can be increased by the addition of a replica victim cache?
2. How to exploit the replicas in either the primary cache or the replica victim cache to provide different levels of reliability and performance?
3. What is the error detection and correction behavior of different replica-based schemes under different soft error rates?

We implemented the proposed replica victim caching schemes by modifying the Simplesclar 3.0 [14]. The error injection experiments are based on *random injection model* [5]. Our experimental results reveal that a replica victim cache of 4 entries can increase the reliability of ICR by 21.7% without impacting performance and its area overhead is less than 10%, compared to most L1 data caches.

The rest of the paper is organized as follows. Section 2 introduces the background information about In-Cache Replication and its limitation. Section 3 describes the architecture of replica victim caching and different schemes to exploit the replica victim lines for improving data reliability. The evaluation methodology is given in section 4. Section 5 presents the experimental results. Finally, section 6 summarizes the contributions of this paper and identifies directions for future research.

2 Background and Motivation

A recent paper [7] presents ICR (In-Cache Replication) for enhancing data cache reliability without significant impact on performance. The idea of ICR is to replicate hot cache lines in active use within the dead cache lines. The mapping between the primary data and the replica is controlled by a simple function. Two straightforward mapping functions are studies in [7], namely, the vertical mapping (replication across sets) and the horizontal mapping (replication within the ways of a set) [7] , as shown in figure 1. The dead cache lines are predicted by a time-based dead block predictor, which is shown to be highly accurate and has low hardware overhead [8]. The ICR approach can be used either with parity or ECC based schemes and there is a large design space to explore, including what, when and where to replicate the data, etc [7]. The design decisions adopted in this paper is presented in table 1.

The results in [7] demonstrate that the ICR schemes can achieve better performance and/or reliability than the schemes that use ECC or parity check alone, however, it can only achieve modest reliability improvement due to the limited space it can exploit. Since each L1 data cache has a fixed number of cache lines, and each cache line can be either used to store the primary data to benefit performance or to store the replicas for enhancing data reliability, ICR approaches

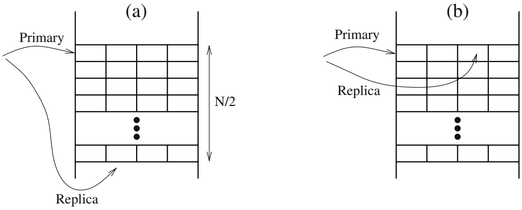
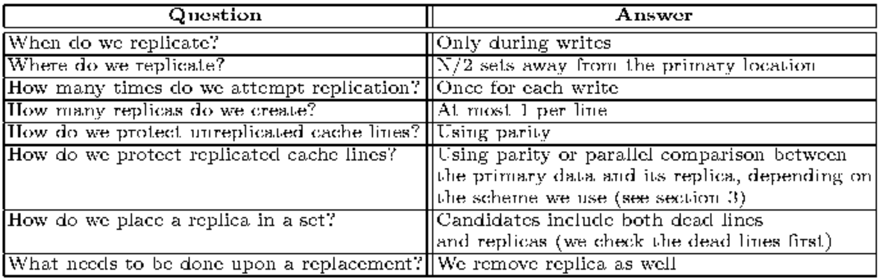


Fig. 1. Cache line replication (a) vertical replication (b) horizontal replication [7].

Table 1. Default implementation strategies regarding cache line replication.



Question	Answer
When do we replicate?	Only during writes
Where do we replicate?	N/2 sets away from the primary location
How many times do we attempt replication?	Once for each write
How many replicas do we create?	At most 1 per line
How do we protect unreplicated cache lines?	Using parity
How do we protect replicated cache lines?	Using parity or parallel comparison between the primary data and its replica, depending on the scheme we use (see section 3)
How do we place a replica in a set?	Candidates include both dead lines and replicas (we check the dead lines first)
What needs to be done upon a replacement?	We remove replica as well

give priority to performance by using several strategies. Firstly, the dead block predictor is designed to be very accurate so that the blocks predicted dead will most likely not be accessed in the near future. Otherwise, the primary data has to be loaded from the higher level memory hierarchy, resulting in performance penalty. Secondly, in case of a conflict between the primary data and the replica, ICR gives higher priority to the primary data and the replica is simply discarded. For instance, when a primary data is written to a cache block, which stores a replica for another primary data, the replica will be overwritten by the coming primary data. As a result, ICR approach can only the replicas that do not conflict with the primary data, resulting in moderate data reliability improvement. The experiments in [7] also reveal that 10% to 35% of load hits in L1 cannot find their replicas and ICR schemes have more than 20% unrecoverable loads under intense error injection. With the trend of increasing soft error rate, the reliability of ICR approaches need to be improved further, especially for applications that demand very high data reliability or operate under highly-noisy environments.

3 Replica Victim Cache

While one straightforward way to enhance data reliability of ICR further is to make more replicas in the L1 data cache by giving priority to replicas in case of their conflicts with the primary data. This approach, however, can inevitably degrade performance and thus is not acceptable. Another approach used in mission-critical applications is the NMR (N Modular Redundancy) scheme, which replicate the data cache for multiple times. However, the NMR scheme is too costly for microprocessors or embedded system with cost and area constraint.

This paper proposes an approach to enhance data reliability of ICR without performance degradation or significant area overheads. The idea is to add a small fully-associative replica victim cache to store the replica victim lines whenever they are conflicting with primary cache lines. Due to the fully associativity of the replica victim cache and data locality (replicas also exhibit temperal and spatial locality, since they are the “images” of the primary data), a very large percentage of load hits in the L1 can find their replicas available either in the L1 data cache or in the replica victim cache.

Victim cache is not a new idea in cache design. Jouppi proposed the victim cache to store the victim lines evicted from the L1 cache for reducing the conflict misses [10]. However, the victim cache proposed by Jouppi cannot be used to enhance data reliability, since there is no redundant copies in both the L1 cache and the victim cache (i.e., only the blocks evicted from the L1 are stored in the victim cache). While the original victim cache aims at performance enhancement, the objective of the replica victim cache is to improve data integrity of ICR approaches significantly without impacting performance. The replica victim cache is a small fully associative cache in parallel with the L1 data cache, as shown in figure 2. In addition to In-Cache Replication, the replica victim cache is used to store replicas for the primary data in the L1 in the following cases:

1. There is no dead block available in the L1 data cache to store the replica.

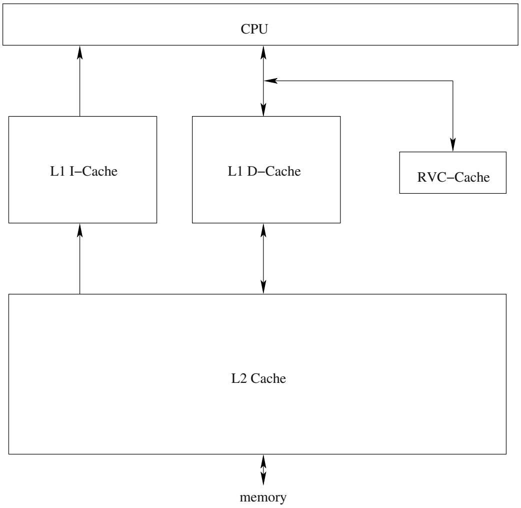


Fig. 2. The architecture of replica victim cache.

2. The replica is replaced by the primary data since ICR gives priority to the primary data.
3. The replica is replaced by another replica for another primary data (note that for a set-associative data cache, multiple replicas can be mapped to the same dead block with ICR approach [7]).

Since ECC computation has performance overhead, we assume all the cache blocks of the L1 and the replica victim cache are protected by parity check. The replicas (both in the replica victim cache and the L1 data cache due to ICR) can be read at each read hits in the L1 for parallel comparison with the primary data to detect multiple bit errors. Alternatively, the replicas can be read only when the parity bit of the primary data indicates an error. The former scheme can enhance data reliability greatly, but there is a performance penalty for the parallel comparison. We assume it takes 1 extra cycle to compare the primary data and the replica in our simulations. The latter scheme also improve the reliability by being able to recover from single bit errors.

The paper examines the following problems.

1. How does the reliability, in terms of loads with replica (see 4), relate to the size of the replica victim cache, the size and associativity of the primary cache? How much loads with replica can be increased by the addition of a replica victim cache?

2. How to exploit the replicas in either the primary cache or the replica victim cache to provide different levels of reliability and performance?

3. How does the error detection and correction behavior of different replica-based schemes under different soft error rates?

To answer the above questions, we propose and evaluate the following schemes:

- `BaseP`: This is a normal L1 data cache without the replica victim cache. All cache blocks are protected by parity. The load and store operations are modeled to take 1 cycle in our experiments.
- `BaseECC`: This scheme is similar to BaseP scheme except that all cache blocks are protected by ECC. Store operations still take 1 cycle (as the writes are buffered), but loads take 2 cycles to account for the ECC verification.
- `RVC-P`: This scheme implements the In-Cache Replication and the proposed replica victim caching. When there is a conflict between the primary data and the replica, the replica victim is stored to the replica victim cache. If the replica victim cache is full, the least-recently used replica is discarded. All cache blocks are protected by parity and the replica is only checked if the parity bit of the primary data indicates an error. Load and store operations are modeled to take 1 cycle in our experiments.
- `RVC-C`: This scheme is similar to RVC-P scheme except that the replica is compared with the primary data before the load returns. The search of replica can be executed simultaneously in both the L1 data cache and the replica victim cache. If the replica hits in the L1 data cache, that replica is used to compare with the primary data; otherwise, the replica in the replica victim cache is used for comparison if there is a hit in the replica victim cache. Note that we give priority to the replicas found in the L1 data cache, because the L1 contains the most updated replicas while the replica victim cache may not (because it only contains the most updated replica victims). However, for the given primary data, if its replica cannot be found in the L1 data cache, the replica found in the replica victim cache must contain the most updated value because every replica victim of the data must have been written to the replica victim cache. We conservatively assume that the load operations take 2 cycles, and store operations take 1 cycle as usual.

It should be noted that in addition to parity check and ECC, the write-through cache and speculative ECC loads are also widely employed for improving data reliability. For write-through caches, data redundancy is provided by propagating every write into the L2 cache. However, write-through caches increase the number of writes to the L2 dramatically, resulting in the increase of write stalls even with a write-buffer. Thus both performance and energy consumption can be impacted [7]. Another way to improve data reliability while circumventing the ECC time cost is the speculative ECC load scheme, which performs the ECC checks in the background while data is loaded and the computation is allowed to proceed speculatively. While speculative ECC loads can potentially hide the access latency, it is difficult to stop the error propagation in a timely manner and may result in high error recovery cost. Since ECC computation consumes more energy than parity check, it is also shown that speculative ECC load has worse energy behavior than the ICR approach that uses parity check only (i.e., ICR-P-PS) [7]. Due to these reasons, we focus on investigating approaches to improve reliability for write-back data caches, and we only compare our approach directly with the recently-proposed ICR approaches, which have exhibited better performance and/or energy behavior than the write-through L1 data cache and the speculative ECC load [7].

4 Evaluation Methodology

4.1 Evaluation Metrics

To compare performance and reliability of different schemes, we mainly use the following two metrics:

- *Execution Cycles* is the time taken to execute 200 million application instructions.
- *Loads with Replica* is the metric proposed in [7] to evaluate the reliability enhancement for data caches. A higher loads with replica indicates higher data reliability, as illustrated by error injection experiments [7]. Since we add a replica victim cache to the conventional L1 data cache architecture, we modify the definition of loads with replica proposed in [7] to be the fraction of read hits that also find their replicas either in the L1 data cache or in the replica victim cache. Note that the difference between our definition and the definition in [7] is that in our scheme, the replica of "dirty" data can be found either in the L1 data cache or in the replica victim cache; while in the ICR scheme [7], the replicas can be only found in the L1 data cache.

4.2 Configuration and Benchmarks

We have implemented the proposed replica victim caching schemes by modifying the Simplesclar 3.0 [14]. We conduct detailed cycle level simulations with sim-outorder to model a multiple issue superscalar processor with a small fully-associative replica victim cache. The default simulation values used in our experiments are listed in Table 2 (note that we do not list the configuration of the replica victim cache in Table 2, since we need to make experiments with different replica cache size).

We select ten applications from the SPEC 2000 suite [16] for this evaluation. Since the simulations of these applications are extremely time consuming, we fast forward the first half billion instructions and present results for the next 200 million instructions. Important performance characteristics of these applications in the base scheme are given in Table 3.

5 Results

5.1 The Size of the Replica Victim Cache and Data Integrity Improvement

Our first experiment is to investigate what is the appropriate size for the replica victim cache. On one hand, the replica victim cache must be small to minimize

Table 2. Configuration parameters in our base configuration for a superscalar architecture. All caches are write-back.

Configuration Parameter	Value
Processor	
Functional Units	4 IALUs, 4FPU
LSQ Size	8 Instructions
RUU Size	16 Instructions
Fetch Width	4 instructions/cycle
Decode Width	4 instructions/cycle
Issue Width	4 instructions/cycle
Commit Width	4 instructions/cycle
Fetch Queue Size	4 instructions
Cycle Time	1ns
Cache and Memory Hierarchy	
L1 Instruction Cache	16KB, 1-way, 32 byte blocks, 1 cycle latency
L1 Data Cache	16KB, 4-way, 32 byte blocks, 1 cycle latency
L2	256K, 4-way, 64 byte blocks, 6 cycle latency
Memory	100 cycle latency

Table 3. Benchmarks from SPEC2000. The last column give the execution cycles for Base scheme.

Benchmark Name	Description	Number of Data References	Number of Cache Misses	Execution Cycles of Base
164.gzip	Compression	58582206	1167237	129215053
175.vpr	FPGA circuit placement and routing	87602536	1968330	202606221
176.gcc	C programming language compiler	79860452	982483	220017658
181.mcf	Combinational optimization	112113406	10423182	240140816
255.vortex	Object-oriented database	110330626	1591461	231793925
256.bzip2	Compression	96251361	1550219	134186684
177.mesa	3D graphics library	98933099	224339	199721006
179.art	C Image recognition/neural networks	87569639	7144640	236659516
183.equake	Seismic wave propagation simulation	64742897	373522	118643743
188.ammp	C Computational chemistry	118184707	15826817	354806058

the hardware overheads. On the other hand, the replica victim cache should be large enough to accommodate the replica victims as many as possible. Due to data locality, it is possible to use a small replica victim cache to store the replica victims for the data, which is most likely accessed in the future. We use an empirical approach to find the best size for the replica victim cache. Specifically, we make experiments on two randomly selected benchmarks (i.e., bzip2 and equake) for replica victim caches with different sizes varying from 1 block to 16 blocks and the L1 data cache is fixed to be 16K-Byte, 4-way set associative, as given in table 2. The loads with replica results are presented in figure 3. As can be seen, the loads with replica increases dramatically when the size of the replica victim cache is increased from 1 block to 2 blocks, because the conflicts of replicas in the replica victim cache can be reduced by exploiting the associativity. For the replica victim cache of 4 or more blocks, the loads with replica is larger than 98.6%, which is tremendously larger than the loads with replica achieved by ICR schemes. We use cacti 3.2 model [15] to estimate the area overhead and the results are shown in table 4. As can be seen, the area overhead of the 4-entry replica victim cache is less than 10%, compared to a data cache of 16K, 32K or 64K bytes. Considering both reliability enhancement and hardware overhead, we fix the size of the replica victim cache to be 4 entries.

Table 4. The area of a 4-entry fully associative replica victim cache, compared with 4-way associative L1 data caches of 16K, 32K and 64K bytes respectively. The cache block size of both the replica victim cache and the data cache are 32 bytes.

	128B RVC	16KB D-cache	32KB D-cache	64KB D-cache
area(cm^2)	0.001183	0.011899	0.021325	0.038569
ratio	100%	9.94%	5.55%	3.07%

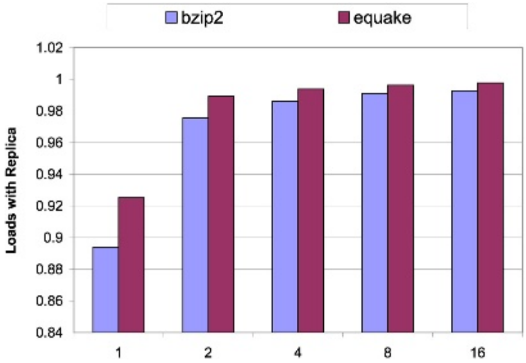


Fig. 3. Loads with replica for replica victim caches of 1, 2, 4, 8 and 16 blocks.

Figure 4 illustrates the loads with replica for a replica victim cache with 4 blocks for all the 10 benchmarks. The replicas can be found either from the L1 data cache (as the ICR approach) or from the replica victim cache. We find that for each benchmark, replica victim cache can store a large portion of the replica victims that are most likely accessed in the future, in addition to the replicas produced by the ICR approach, resulting in significant enhancement on data reliability. The average loads with replica with the replica victim cache is 94.4%, which is 21.7% larger than the ICR approach alone.

5.2 Performance Comparison

Using the above settings for the replica victim cache, we next study the performance of the replica victim caching. Since the replica victim cache is only used to store the replica victims from the L1 data cache, there is no performance degradation compared to the corresponding ICR approaches [7]. Therefore, we only compare the performance implications of the four schemes described in Section 3.

As shown in figure 5, the performance of the RVC-P scheme is comparable to the BaseP scheme and the performance of the RVC-C scheme is comparable to the BaseECC scheme. Specifically, the average performance degradation of RVC-P to BaseP and RVC-C to BaseECC is 1.8% and 1.7% respectively. It should be noted that this performance degradation comes from ICR, not from the replica victim caching. Since ICR relies on dead block prediction and there is no perfect dead block predictor, some cache blocks in the L1 may be predicted

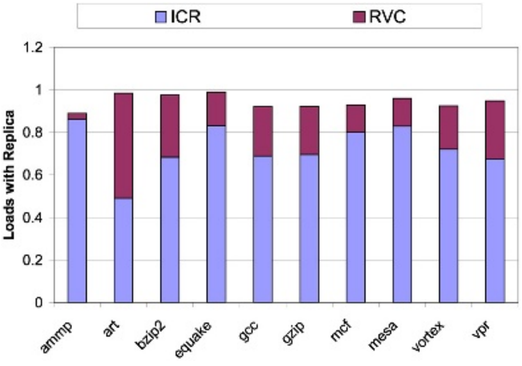


Fig. 4. Loads with replica for replica victim caches of 4 blocks.

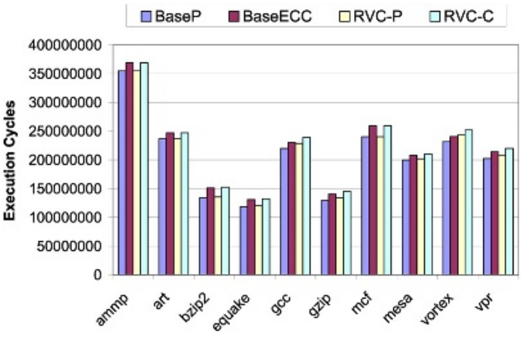


Fig. 5. Performance comparison of different schemes.

dead (and thus are utilized to store replicas) but are actually accessed later (i.e., not "dead" yet), which can result in performance degradation.

5.3 Error Injection Results

We conduct the error injection experiment based on *random injection model* [5]. In this model, an error is injected in a random bit of a random word in the L1 data cache. Such errors are injected at each clock cycle based on a constant probability (called error injection rate in this paper).

Figure 6 and figure 7 present the fraction of loads that could not be recovered from errors (including single-bit or mult-bit errors) for BaseP, BaseECC, ICR-P-PS, RVC-P and RVC-C for bzip2 and vpr respectively. In both experiments, the data is loaded as a function of the probability of an error occurring in each cycle and the error injection rate varies from 1/100 to 1/1000 and 1/10000. Note that the intention here is to study the data reliability under intense error behavior, thus very high error injection rates are used. As can be seen, BaseP always has the worst error resilient behavior, since the parity bit can only detect single-bit errors. When the error injection rate is relatively low (i.e., 1/10000), BaseECC has similar percentage of unrecoverable loads as RVC-C. However, as the error injection rate increases, the difference between BaseECC and RVC-C grows larger. Specifically, when the error injection rate is 1/1000, RVC-C can reduce the unrecoverable load by 9.1% and 4.5% for bzip2 and vpr respectively, compared to the BaseECC scheme. Similarly, the RVC-P scheme exhibits much better error resilient behavior compared to BaseP and ICR-P-PS at different error injection rate.

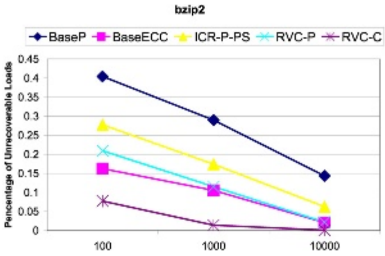


Fig. 6. Percentage of unrecoverable loads for bzip2.

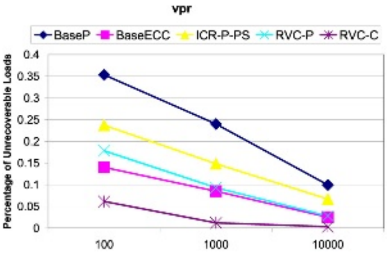


Fig. 7. Percentage of unrecoverable loads for vpr.

5.4 Sensitivity Analysis

To verify the effectiveness of the replica victim cache of 4 blocks for L1 data caches with different configurations, we also make experiments to study the loads with replica by varying the L1 data cache size and the number of associativity. In both experiments, the replica victim cache is fixed to be a fully-associative cache of 4 blocks.

Figure 8 gives the loads with replica for L1 data caches of 8K, 16K, 32K, 64K and 128K bytes. The block size and the associativity of the L1 data cache are 32 bytes and 4 way respectively. The results are very interesting. As can be seen, bzip2 and equake exhibit different trends in loads with replica. As the data cache size increases, the loads with replica of bzip2 decreases slightly, while the loads with replica of equake increases slightly. The reason is that replicas can be found in two places: the L1 data cache and the replica victim cache. The number of replicas that can be stored in the L1 increases as the L1 data cache size increases, however, the relative number of replicas that can be stored in the replica victim cache decreases since the size of replica victim cache is fixed. Therefore, the effect of increasing the L1 data cache size on the loads with replica is dependent on these two factors. The breakdown of loads with replica from the L1 data cache and from the replica victim cache for bzip2 and equake are presented in figure 9 and figure 10 respectively. In figure 9, the decrease of loads with replica from the replica victim cache dominates, and thus the total loads with replica decreases as the L1 data cache size increases. In contrast, in figure 10, the increase of loads with replica from the L1 data cache dominates, and hence the total loads with replica increases with the increase of the L1 data cache size. However, for all the L1 data cache configurations, the replica victim cache of 4 entries can achieve the loads with replica more than 98.1% on average, which is substantially larger than what the ICR approach alone can achieve.

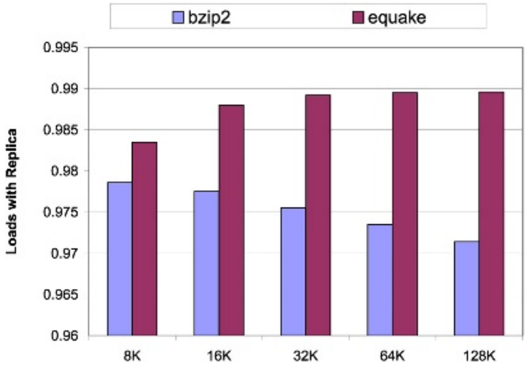


Fig. 8. Loads with replica for 4-way associative L1 data caches of 8K, 16K, 32K, 64K and 128K bytes. The replica victim cache is fully associative with 4 blocks. The block size of both the L1 data cache and the replica victim cache is 32 bytes. Insensitive to the L1 data cache size, the addition of the fully associative replica victim cache of 4 blocks can achieve the loads with replica more than 98.1% on average.

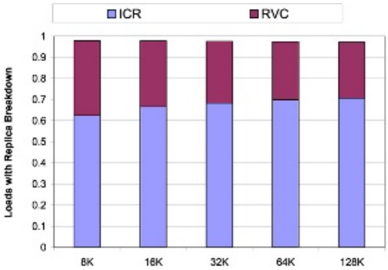


Fig. 9. Loads with replica breakdown for L1 data caches of 8K, 16K, 32K, 64K and 128K for bzip2.

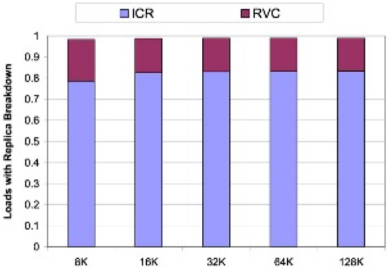


Fig. 10. Loads with replica breakdown for L1 data caches of 8K, 16K, 32K, 64K and 128K for equake.

We also study the loads with replica for L1 data caches with different associativity and find similar results. Therefore, with the addition of a small fully-associative replica victim cache of 4 entries, a very high loads with replica can be achieved to enhance data reliability of ICR further for a variety of L1 data caches.

6 Conclusion

This paper studies the limitation of In-Cache Replication and proposes to add a small fully-associative replica victim cache to store the replica victim lines in case of their conflicts with the primary data. We find that with the addition of a small replica victim cache of 4 entries, the loads with replica of the ICR scheme can be increased by 21.7%. On average, 94.4% load hits in L1 can find replicas either in the L1 data cache or in the replica victim cache. We also propose and evaluate two different reliability enhancing schemes — RVC-P and RVC-C — that are proven to be quite useful.

RVC-P is a much better alternative for ICR-P-PS where one may want simple parity protection. It can enhance reliability significantly by providing additional replicas in the replica victim cache without compromising performance. RVC-P also has better performance than RVC-C or ECC based schemes (i.e., BaseECC).

RVC-C can increase the error detection/correction capability by comparing the primary data and the replica before the load returns. Our error injection experiments reveal that RVC-C has the best reliability and can be used for applications that demand very high reliability or operate under highly noisy environments. Compared with the BaseECC scheme, the performance degradation of RVC-C is only 1.7% on average.

In summary, this paper proposes the addition of a small fully-associative replica victim cache to enhance data reliability of ICR schemes significantly without compromising performance. Our future work will concentrate on studying the reliability and performance impact of replica victim cache for multiprogramming workloads. In addition, we plan to investigate how to use a unified victim cache efficiently to store both primary victims and replica victims and the possibility to make a better tradeoff between performance and reliability.

References

1. P. Sweazey. SRAM organization, control, and speed, and their effect on cache memory design. Midcon/87, pages 434-437, Septembe, 1987.
2. H. Imai. Essentials of error-control coding techniques. Academic Press, San Diego, CA, 1990.
3. C.L. Chen and M.Y Hsiao. Error-correcting codes for semiconductor memory applications: a state of the art review. In Reliable Computer Systems - Design and Evaluation, pages 771-786, Digital Press, 2nd edition, 1992.
4. J. Karlsson, P. Ledan, P. Dahlgren, and R. Johansson. Using heavy-ion radiation to validate fault handling mechanisms. IEEE Micro, 14(1):8–23, February 1994.
5. S. Kim and A. Somani. Area efficient architectures for information integrity checking in cache memories. ISCA, May 1999, pp. 246–256.
6. J. Sosnowski. Transient fault tolerance in digital systems. *IEEE Micro,* 14(1):24–35, February 1994.
7. W. Zhang, S. Gurumurthi, M. kandemir and A. Sivasubramaniam. ICR: in-cache replication for enhancing data cache reliability, DSN, 2003.
8. S. Kaxiras, Z. Hu, and M. Martonosi. Cache decay: exploiting generational behaviour to reduce cache leakage power, ISCA, June 2001.

9. P.Shivakumar, M. Kistler, S. Keckler, D. Burger and L. Alvisi. Modeling the effect of technology trends on soft error rate of combinational logic, DSN, June, 2002.
10. N.P. Jouppi. Improving direct-mapped cache performance by the audition of a small fully-associative cache and prefetch buffers, ISCA, 1990.
11. M. Hamada and E. Fujiwara. A class of error control codes for byte organized memory system-SbEC-(Sb+S)ED codes. IEEE Trans. on Computers, 46(1):105-110, January 1997.
12. S. Park and B. Bose. Burst asymmetric/unidirectional error correcting/detecting codes, FTC, June, 1990.
13. Understanding Soft and Firm Errors in Semiconductor Devices. Actel Whitepaper, 2002.
14. http://www.simplescalar.com.
15. S. Wilton and N. Jouppi. CACTI: An enhanced cache access and cycle time model. IEEE Journal of Solid-State Circuits, Vol. 31(5):677-688, May 1996.
16. http://www.spec.org.

Efficient Victim Mechanism on Sector Cache Organization

Chunrong Lai [1] and Shih-Lien Lu [2]

[1] Intel China Research Center, 8F, Raycom Infotech Park A, No.2 Kexueyuan South Road
ZhongGuanCun, Haidian District, Beijing China, 100080
chunrong.lai@intel.com

[2] Microprocessor Research, Intel Labs
shih-lien.l.lu@intel.com

Abstract. In this paper we present an victim cache design for caches organized with line that contains multiple sectors (sector cache). Sector caches use less memory bits to store tags than non-sectored caches. Victim cache has been proposed to alleviate conflict misses in a lower associative cache design. This paper examines how victim cache can be implemented in a sector cache and proposes a further optimization of the victim buffer design in which only the tags of the victim lines are remembered to re-use data in the sector cache. This design is more efficient because only an additional "OR" operation is needed in the tag checking critical path. We use a full system simulator to generate traces and a cache simulator to compare the miss ratios of different victim cache designs in sector caches. Simulation results show that this proposed design has comparable miss ratios with designs having much more complexity.

1 Introduction

In a cache an address tag (or tag) is used to identify the memory unit stored in the cache. The size of this memory unit affects how well a cache functions. For a fixed size cache larger unit size needs less memory bits to store tags and helps programs that possess special locality. However, larger unit may cause fragmentation making the cache less efficient when spatial locally is not there. Moreover, transferring each unit from lower memory hierarchy takes higher bandwidth. Smaller unit size allows more units to be included and may help programs that spread memory usage.

Sector cache[1][2] has been proposed as an alternative to strike a balance of cache unit sizes. A sector cache's memory unit is divided into sub-sections. Each unit needs only one tag thus saves tag memory bits. These sub-sections of a sector cache need not to be simultaneously brought in the cache allowing lower transferring bandwidth. Another advantage of sector caches is observed for multiprocessors systems because they reduce false sharing[3][4]. Sector cache's advantage is evident in that many microprocessors employ sector caches in their designs. For example, Intel's Pen-

P.-C. Yew and J. Xue (Eds.): ACSAC 2004, LNCS 3189, pp. 16–29, 2004.

tium® 4[1][5], SUN's SPARC™[6] and IBM's POWERPC™ G4™[7]/G5™[8] all employ sector cache in their cache organization.

This work intends to propose and evaluate further optimization techniques to improve performance of a sector cache. One of those designs is the victim cache[9]. A victim cache includes an additional victim buffer. When a line is replaced it is put into this small buffer which is full associative instead of just being discarded. The idea is to share the victim buffer entries among all sets since only a few of them are hotly contended usually. First, we discuss how victim buffer/cache idea can be applied in a sector cache. We evaluate two implementations of victim cache. One is called "line-victim" and the other is "sector-victim". We further propose a third victim mechanism design named "victim sector tag buffer"(VST buffer) for further utilize the sector cache lines. This design tries to address a sector cache's potential disadvantage of having larger unit size and could be under-utilized.

Since there are many different names[3][6][10][11][12][13][14][15] used to describe the units used in a sector cache, we first describe the terminology used in this paper. In our terminology a cache consist of lines which have tags associated with each of them. Each line consists of sub-units which are called sectors. This naming convention is the same as described in the manuals of Pentium® 4[5] and POWERPC™[7][8]. An example 4-way set-associative cache set is shown in figure 1. A valid bit is added to every sector to identify a partial valid cache line. We also use the terminology s-ratio which is defined as the ratio between the line size and the sector size. A sector cache with s-ratio equals to p is called p-sectored cache as [1]. The example in figure 1 it is a 4-sectored cache.

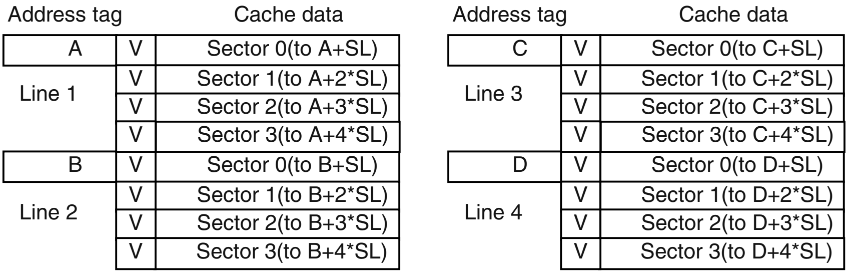


Fig. 1. Principles of sectored cache

This paper is organized as follows. In this section we introduce the concept of sector cache and victim mechanism. In the next section we first review other related works in this area. We then describe in more detail of our design. In section three we present the simulation methodology. In section four and five we introduce our simulation results on different cache levels. Finally we conclude with some observations.

[1] Pentium is a registered trademark of Intel Corp. or its subsidiaries in the United States and other countries.

2 Sector Cache with Victim Buffer

2.1 Related Work

Sector caches can be used to reduce bus traffic with only a small increase in miss ratio[15]. Sector cache can benefit in two-level cache systems in which tags of the second level cache are placed at the first level, thus permitting small tag storage to control a large cache. Sector cache also is able to improve single level cache system performance in some cases, particularly if the cache is small, the line size is small or the miss penalty is small. The main drawback, cache space underutilization is also shown in [13].

Rothman propose "sector pool" for cache space underutilization[13]. In the design, each set of set-associative cache compose of totally s-ratio sector lists. Each list has a fix number of sectors that the number is less than the associativity. S-ratio additional pointer bits, associate with a line tag, point to the actual sector as the index of the sector list. Thus a physical sector can be shared in different cache lines to make the cache space more efficient. Unlike our victim mechanism who tries to reduce the cache Miss ratio, this design more focus on cache space reduction. It depends on a high degree set associative cache. The additional pointer bits and the sector lists will make the control more complex. For example, the output of tag comparison need to be used to get the respond pointer bit first then can get the result sector. This lengthens the critical path. Another example is that different replacement algorithms for the cache lines and sector list need to be employed at the same time.

Seznec propose "decoupled sectored cache"[1][11]. A [N,P] decoupled sectored cache means that in this P-sectored cache there exists a number N such that for any cache sector, the address tag associated with it is dynamically chosen among N possible address tags. Thus a log2N bits tag, known as the selection tag, is associated with each cache sector in order to allow it to retrieve its address tag. This design increases the cache performance by allow N memory lines share a cache line if they use different sectors that some of the sectors have to be invalid at normal sector cache design. Our concern about this design is that the additional tag storage, say N-1 address tags and s-ratio * log2N selection tags for each line, need large amount of extra storage. Seznec himself use large(32 or 64) s-raio, which will make the validity check and coherence control very complex, to reduce tag storage before decoupling. We tried to implement this idea and saw the line-fill-in and line-replacement policy is important for the performance. If with a line-based-LRU-like fill-in/replacement policy proposed by ourselves, since Seznec did not give enough details of his policies, the decoupled sector cache will not perform better than our VST design, if with similar extra storage, given s-ratio range of 2~8. And, an additional compare need to be performed to the retrieved selection tag to ensure the sector data is right corresponded to the address tag which causes the tag matching. This also lengthens the tag checking critical path.

Victim caching was proposed by Jouppi[9] as an approach to reduce the cache Miss ratio in a low associative cache. This approach augments the main cache with a small fully-associate victim cache that stores cache blocks evicted from the main cache as a result of replacements. Victim cache is effective. Depending on the program, a four-entry victim cache might remove one quarter of the misses in a 4-KB direct-mapped data cache. Recently [16] shows that victim cache is the most power/energy efficient design among the general cache misses reduction mechanisms. Thus it becomes a more attractive design because of the increasing demand of low power micro-architecture.

2.2 Proposed Design

In order to make our description clearer, we define several terms here. We call a reference to a sector cache a *block-miss* if the reference finds no matching tag within the selected set. We call a reference a *data-miss* if the tag matches but the referenced word is in an invalid sector. Thus a *miss* can be either block-miss or data-miss. Similar to [1][11] describe, for a P-sectored cache we divide the address A of a memory block in four sub-strings (A3, A2, A1, A0) defined as: A0 is a log2SL bit string which SL is the sector length, A1 is a log2P bit string show which sector this block is in if it is in a cache line, A2 is a log2nS bit string which nS is the number of the cache sets, A3 consists of the remaining highest significant bits. The main cache line need store only the bits in A3. Figure 2 show tag checking of directed-mapped case. A2 identify the only position of the tag to be compared in the tag array. (A2, A1) identify the only position the data sector can be. The data can be fetched without any dependency on the tag comparison. The processor pipeline may even start consuming the data speculatively without waiting for the tag comparison result, only roll back and restart with the correct data in the case of cache miss which is rarely happened, as line prediction.

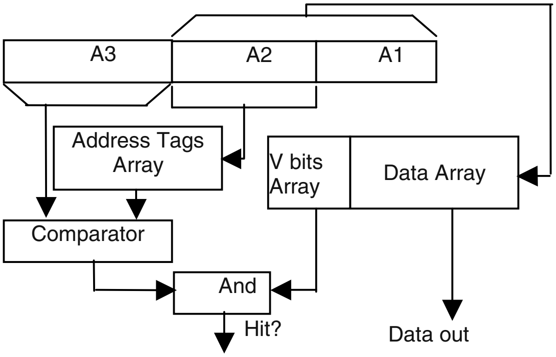


Fig. 2. Directed mapped sector cache tag checking

In the case of set-associative cache, (A2, A1) can only select the conceptual "sector set", then waiting for the comparison result of the address tags to get a line ID to deliver the correspond sector. Figure 3 is such an example of a 2-way associate sector cache. In a lower-associate cache the sector data and the valid bits being select can be got independently with the tag comparison.

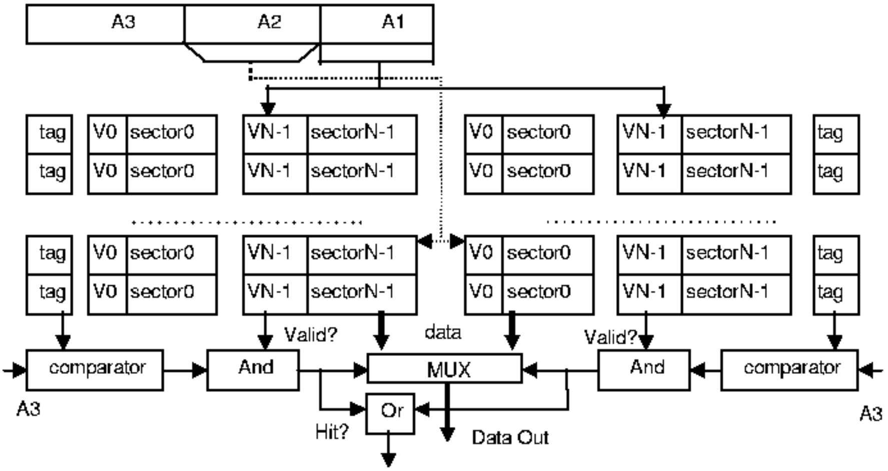


Fig. 3. 2-way associate sector cache tag checking

In figure 3 a line ID is needed, in critical path, as selection signal of the MUX. Line ID is founded after the tag comparison result. For a higher associate cache like a CAM, where a simple MUX may not be used, the data and valid bits could be not right at hand immediately. But Line ID retrieving still dominates in the critical path there[17].

As mentioned by many researchers victim cache can reduce cache Miss ratio. There are two straightforward victim designs for sector cache. One is line-victim cache(LVC), the other is sector-victim cache(SVC). Figure4 show their tag checking. Tag checking of the line victim cache is in the left and the other is in the right. The most difference between them is the data unit associate with the victim tag. In line-victim cache, the data unit is a cache line. And the data unit is a sector in the sector-victim cache. Thus the lengths of the victim tags of LVC and SVC are different. For same entries number LVC can be expected more cache misses saved due to more storage there, where SVC can be expected a little faster tag checking and data retrieve. Figure 4 do not connect the victim cache with main cache to avoid unnecessary complexity and allow architects to decide if swap the victim data with the main cache data when hit victim cache.

Both line-victim cache and sector-victim cache are paralleled accessed with the main sector cache. A cache line is evicted to the line- victim cache in case of cache replacement happens. As to the sector-victim cache, only the valid sectors in the whole line are evicted to the sector-victim-cache. Also when a new line is brought into cache, the sector-victim cache is checked to see if there are other sectors in the same line. If so the victim sectors are also brought into the main cache line to maintain a unique position of a cache line.

We know some of the requested data may still be in the data cache but it is just inaccessible because it has been invalidated. This paper describes another approach, called "VST buffer" to remember what is still in the data cache.

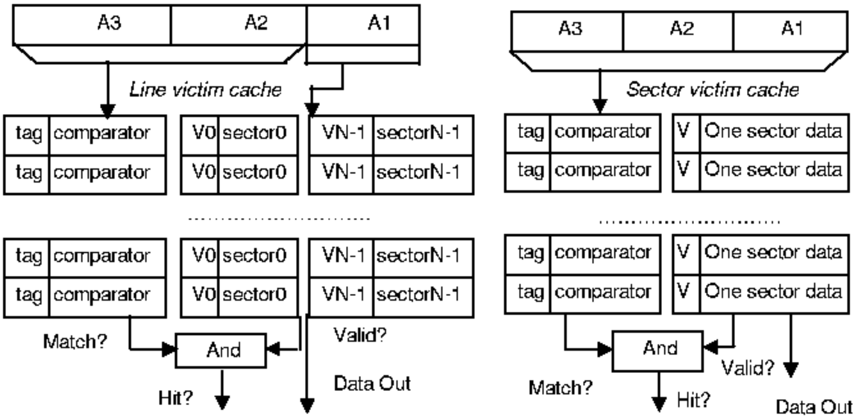


Fig. 4. Tag checking of line-victim cache and sector-victim cache

When a block miss happens and the set is full, a cache line must be replaced. Each sector of the replaced line will be mark as invalid. A new sector will be brought into the replaced line, and the cache tag will be updated. Thus some of the previously replaced line's sector data may still be in the data array since not all sector, of the newly brought in cache line, is brought in. Only their valid bits are marked invalid. VST buffer is used to keep track of these sectors whose data is still the data array. Thus a VST entry consists of the victim tag, the victim valid bits and the "real location" line ID in the cache set. For a directed mapped main cache the line ID field is needless. The left side of figure 5 shows the VST buffer tag checking with a directed-mapped main cache. As seen from the figure the VST buffer produce an additional "VST hit" signal to be perform "or" operation with the main cache hit signal in the critical path, without affecting the sector fetching and consuming. In either a VST hit or a main cache hit the data can be processed continuously.

The right side of figure 5 shows the VST buffer tag checking of a set-associate main cache. A VST hit not only leads to a hit result but also deliver a line ID to the main cache selector to get the result data. This line ID signal is performed "or" operation with the main cache line ID signal to select final line. One extra cost here is when a new sector is brought into the main cache, if a data miss happens, the VST needs to be checked if the position contains a sector being victimized. If so the victim entry is invalidated or thrashed. This does not increase cache-hit latency since it happens when cache miss. Since there is already cache miss penalty the additional cost seems to be acceptable.

When compare the cost of the three victim mechanism connected with a p-Sectored Cache all compose of N entries. We see beside the similar comparators and the control, the line- victim cache need N line tags, data of N * line size and N*P

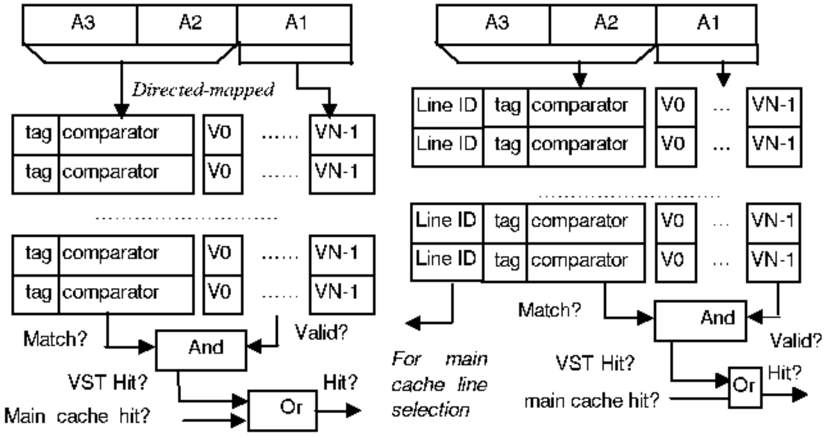


Fig. 5. Tag checking of Victim Sector Tag Buffer (VST buffer) with sector cache

valid bits; the sector-victim cache need N sector tags (each of it is log2P bits longer than a line tag), data of N * sector size and N valid bits; and the VST buffer need N line tags, N * P valid bits and N * log2Assoc bits of line IDs which Assoc is the cache associativity. So the line victim cache needs most resource among them as VST buffer need least resource.

In MP system, where the sector cache is proved efficient, there need additional cache coherence protocol, like MESI, to maintain the cache coherence. We think the victim mechanism will make the MP sector cache coherence protocol more complex. But we will not discuss the details here since it is beyond this paper's scope.

3 Simulation Methodology

Several SpecCPU2K[18] benchmarks (compiled with compiler option "–O3 –Qipo"), Java workload SpecJBB2K[19] with Java runtime environment JSEV1.4.1 which is an integer benchmark, and two commercial-like floating-point benchmarks, one is a speech recognition engine[20], the other is an echo cancellation algorithm[21] in audio signal processing, are used in our study.

In order to consider all the effects, including system calls, we use a full system simulator to generate memory reference traces. The simulator used is called SoftSDV[22]. The host system runs Windows 2000 and the simulated system is WindowsNT in batch mode using solid input captured in files. Then we run the traces through a trace-driven cache simulator.

We generate both L1 memory reference traces and L2 memory reference traces. After 20 billion instructions after system start up (the target application is configured auto-run in the simulation) we collect 200 million memory references as our L1 traces. We use 100 million references of them to warm up L1 cache and analysis the behavior of the latter 100 million. The L1 sector cache we simulated is mainly configured as below with small varieties: 16KBsize, 64B line size, 16B sector size, 4 way associate, LRU replacement algorithm and write-back approach. For L2 cache behavior we use a built-in first level cache together with trace generation. We warm up the built-in cache with 1 billion instructions. Then we collect L2 traces consist of 200 million read references. Also in our simulation we use 50 million L2 references to warm up the L2 cache. The hierarchy consist L2 sector cache we simulated is mainly configured as below with small varieties: L1: 16KBsize, 32B line size, 4-way associate, LRU replacement algorithm and write-back approach. L2: 1MB size, 128 byte line size, 32 byte sector size, 8-way associate, LRU replacement algorithm and write-back approach.

4 Level 1 Sector-Cache Simulation Results and Discussion

We present the L1 simulation data as the Miss Ratio Improvement Percentage (MRIP) of all benchmarks. The reason that we present L1 data first is that it is easier to correlate the observed L1 behavior back with the source code. Figure 6,7 are the MRIP trends with various parameters as the variable. All the numbers are computed as the geometric means of the different workload data also list in the paper. Figure 6 indicates that with larger number of the victim mechanism entries the miss ratio improvement increases. Since VST requires no data array we can implement a much larger victim buffer at the same cost of a smaller SVC/LVC and achieve the same (or even better) performance improvement. For example 128 entries VST performs comparably with 64 entries SVC or 32 entries LVC. Figure 6 also explores the improvement with several sector cache line sizes and sector sizes. We observed that VST performs better with larger s-ratios. This is because of higher underutilization cache space exist with higher s-ratio. On the other hand SVC and LVC performs better with larger line and sector sizes. Figure 7 compares how the victim mechanisms affect caches with different associativities or different cache sizes. It is not surprising to learn that all three forms of victim mechanisms help the lower associative cache better. This is because higher associativity already reduced much of the conflict misses victim cache is targeting. It is also seen smaller L1 cache benefits more from the victim mechanisms. As frequency of microprocessors continues to grow, smaller but faster (lower associativity gives faster cache too) cache will be more prevalent.

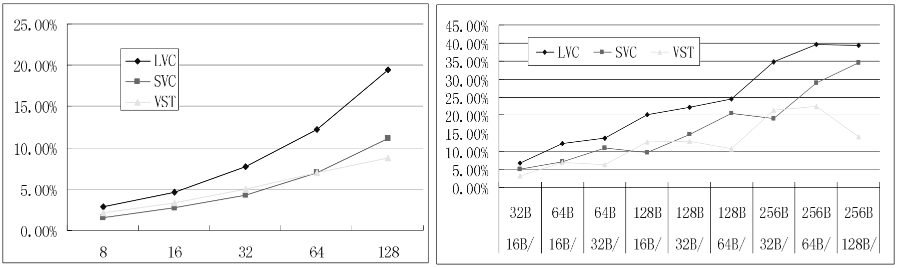


Fig. 6. MRIP with victim entries or line/sector sizes (higher is better)

We observe that LVC gives the best Miss ratio improvement at the highest hardware cost. While the SVC approach we used for this study needs the second highest hardware cost, it is not better than VST approach. The VST approach is a reasonable approach in terms of hardware design complexity and overhead.

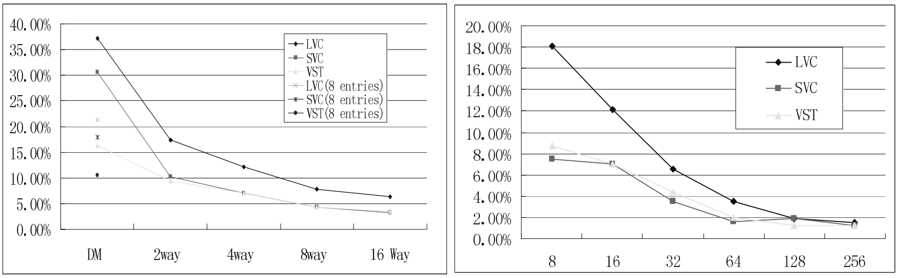


Fig. 7. MRIP with different associativities or L1 sizes

The cache miss ratios with different number of victim entries, correspond to the left figure of figure 6, are listed in table 1. The data of other figures are listed in appendix. Table 1 also list corresponding block misses ratios for further investigation.

Table 1. Miss Ratios and Block Miss Ratios with numbers of victim entries

L1 victim entries		Miss ratios						Block Miss ratios					
		8	16	32	64	128	origin	8	16	32	64	128	origin
LVCS	LVC	2.77	2.73	2.69	2.64	2.57	2.92	1.22	1.20	1.17	1.14	1.10	1.36
	SVC	2.79	2.76	2.73	2.69	2.64	2.92	1.25	1.23	1.20	1.18	1.15	1.36
	VST	2.81	2.78	2.76	2.73	2.71	2.92	1.25	1.23	1.20	1.18	1.16	1.36
AMMP	LVC	9.01	8.94	8.81	8.58	8.25	9.08	3.88	3.85	3.77	3.63	3.42	3.91
	SVC	9.04	9.00	8.93	8.77	8.50	9.08	3.90	3.88	3.85	3.77	3.65	3.91
	VST	9.02	8.96	8.86	8.71	8.56	9.08	3.87	3.83	3.76	3.67	3.57	3.91
MESA	LVC	0.63	0.57	0.56	0.54	0.51	1.67	0.27	0.21	0.20	0.19	0.18	0.95
	SVC	0.72	0.65	0.58	0.55	0.54	1.67	0.35	0.29	0.23	0.20	0.20	0.95
	VST	0.99	1.01	1.02	1.03	1.02	1.67	0.49	0.49	0.49	0.48	0.48	0.95
SAEC	LVC	3.29	3.26	3.19	3.07	2.84	3.35	0.93	0.92	0.90	0.86	0.80	0.95
	SVC	3.34	3.32	3.29	3.25	3.18	3.35	0.94	0.94	0.93	0.92	0.90	0.95
	VST	3.33	3.32	3.30	3.27	3.25	3.35	0.94	0.93	0.92	0.91	0.90	0.95
GZIP	LVC	10.67	10.56	10.37	10.04	9.45	10.81	7.44	7.33	7.13	6.76	6.13	7.58
	SVC	10.68	10.58	10.41	10.12	9.63	10.81	7.47	7.39	7.23	6.97	6.55	7.58
	VST	10.69	10.65	10.45	10.13	9.70	10.81	7.45	7.33	7.12	6.75	6.14	7.58
GCC	LVC	2.32	2.27	2.09	1.76	0.88	2.35	0.68	0.67	0.62	0.53	0.29	0.69
	SVC	2.34	2.33	2.31	2.23	2.02	2.35	0.69	0.68	0.68	0.66	0.60	0.69
	VST	2.30	2.24	2.14	2.07	2.06	2.35	0.67	0.66	0.62	0.60	0.59	0.69
SJBB	LVC	3.88	3.83	3.76	3.67	3.51	3.96	1.63	1.60	1.56	1.52	1.45	1.67
	SVC	3.90	3.88	3.84	3.77	3.69	3.96	1.65	1.63	1.61	1.57	1.53	1.67
	VST	3.91	3.88	3.84	3.78	3.71	3.96	1.65	1.62	1.59	1.55	1.50	1.67

As shown in table 1, the benchmark "mesa" got most of cache misses reduction with victim mechanism regardless LVC/SVC, or VST we used. "ammp" got least misses reduction with LVC and "saec" got least misses reduction with SVC and VST.

For the workload "mesa", we observed the block Miss ratio reduce much more significantly with victim mechanism compared to the cache Miss ratio. Thus with victim mechanism the workload basically keeps more cache lines to save cache misses in this level. Other issues, like quantitative spatial localities that make SVC performs differently, say reduces different percentage of miss ratio reduced by LVC with same entries, play minor role in this level.

In some cases (GCC with 8 victim entries), the VST buffer approach performs better than LVC even without any data array. After investigation we concluded that the VST buffer approach sometimes uses the victim buffer more efficiently and can avoid be thrashed. Victim cache contains data that may be used in future. But the data can also be kicked out of the victim cache before it is needed. For example, streaming accesses, if miss the main cache, will evict main cache lines to update the victim cache. Thus the victim cache gets thrashed and may lost useful information. It plays differently in VST approach. We see in non-sector cache, streaming accesses are mapping in different sets of cache which make it difficult to be detected. In a sector cache the next sector of a cache is inherently subsequence of the previous sector. Figure 8 shows the VST states with one by one streaming accesses (or sequential) going to the cache, only one VST entry is enough handling them since the entry can be re-used(disabled) after a whole main cache line fill-in. Thus the whole buffer will keep longer history. This is right the case VST performs better than LVC for GCC.

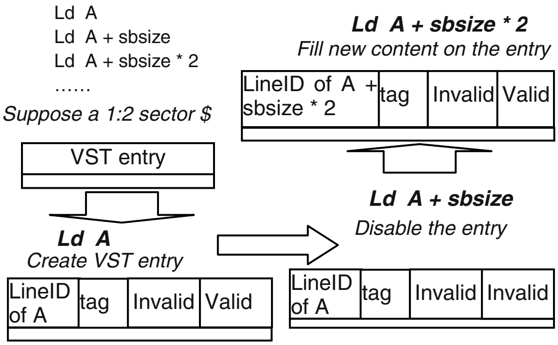


Fig. 8. Avoid be thrashed by streaming access

5 Level 2 Sector-Cache Simulation Results and Discussion

We also explore the possibility of applying our proposed methods on level-two cache design. This time only those references that missed the build-in level one cache are collected in the trace file. Table 2 illustrates the tabulated result in terms of miss ratio for various entries. Data with other parameters are also listed in appendix.

There are several observations made from the L2 data. First, LVC performs better than SVC with same entries but worse than SVC with s-ratios, here 4 times, of entries, same as be observed from L1 data. Second, in lower level set-associative cache, victim mechanism performs differently as L1. It does not save so many cache misses as

Table 2. L2 Miss Ratio with Victim Mechanism

Entries		16	32	64	128	256	Origin
Ammp	LVC	15.36	15.06	14.52	13.71	12.85	15.62
	SVC	15.49	15.34	15.06	14.58	13.89	15.62
	VST	14.82	14.24	13.71	13.30	12.91	15.62
LVCSR	LVC	36.57	36.53	36.46	36.33	36.06	36.60
	SVC	36.59	36.57	36.54	36.49	36.38	36.60
	VST	36.57	36.54	36.48	36.36	36.14	36.60
Mesa	LVC	9.64	9.64	9.63	9.63	9.63	9.64
	SVC	9.64	9.64	9.64	9.64	9.63	9.64
	VST	9.64	9.64	9.64	9.64	9.64	9.64
Gcc	LVC	8.52	8.51	8.50	8.48	8.44	8.52
	SVC	8.52	8.52	8.51	8.50	8.49	8.52
	VST	8.51	8.51	8.49	8.47	8.45	8.52
Gzip	LVC	0.33	0.33	0.33	0.33	0.33	0.33
	SVC	0.33	0.33	0.33	0.33	0.33	0.33
	VST	0.33	0.33	0.33	0.33	0.33	0.33
Mcf	LVC	50.47	50.43	50.36	50.23	49.98	50.50
	SVC	50.48	50.47	50.44	50.38	50.26	50.50
	VST	50.47	50.45	50.40	50.35	50.24	50.50
SAEC	LVC	0.40	0.38	0.37	0.37	0.37	0.41
	SVC	0.41	0.40	0.39	0.38	0.37	0.41
	VST	0.42	0.42	0.42	0.42	0.42	0.41

L1 cache. This is not surprising since a small L1 already catch a significant part of data locality and L2 reference patterns tend to be more irregular. Third, the VST buffer performs well among the three victim mechanisms in this memory hierarchy level. It can outperform LVC and SVC for the benchmark “ammp”. Even it is more difficult to correlate the L2 references back with the source or binary, than L1 references. We still ascribe the better VST performance to its property of avoiding be thrashed. As to the workloads, “ammp” and “SAEC” get most significant cache misses reduction here. This behavior is opposite to the L1 behavior. Also the significant block miss reduction can not be observed in this level as the data in appendix shows. Thus we suggest that the extra storage of LVC and SVC benefit more from the general data locality; and VST benefit more from the cache underutilization whether the reference pattern is regular or not.

6 Conclusion

We have described three possible implementation of victim buffer design in a sector cache. They have different complexity and hardware overhead. Several up-to-date applications are used to evaluate their performance in terms of miss ratio. Overall three mechanisms have comparable cache misses reduction. For a directed-mapped Level 1 cache, the mechanisms can save significant amount of cache misses.

Among the three mechanisms LVC gives the best performance with highest overhead. Whether SVC is performance/cost effective or not rely on the quantitative spatial locality of the workload.

We also investigate several benefits of VST in this paper. Include the low-cost design, keeping longer victim history and be more able to capture irregular reference pattern in lower memory hierarchy.

Acknowledgement. We thank the AudioProcessing group and the OpenRuntimePlatform group of Intel China Research center for giving us their up-to-date workloads and providing helpful discussions on porting workloads to our simulator. We also thank Zhu Ning and Peter Liou for providing necessary computing infrastructure support.

References

[1] Andre. Seznec. "Decoupled sectored caches". IEEE Trans. on Computers, February, 1997

[2] D.A.Patterson, J.L.Hennessy, "Computer architecture: A quantitative approach", Morgan Kaufmann Publishers Inc., San Francisco,1996.

[3] Kuang-Chih Liu, Chung-Ta King, "On the effectiveness of sectored caches in reducing false sharing misses" International Conference on Parallel and Distributed Systems, 1997

[4] Won-Kee Hong, Tack-Don Han, Shin-Dug Kim and Sung-Bong Yang, "An Effective Full-Map Directory Scheme for the Sectored Caches", International Conference/Exhibition on High Performance Computing in Asia Pacific Region, 1997

[5] Hinton, G; Sager, D.; Upton, M.; Boggs, D.; Carmean, D.; Kyker, A.; Roussel, P., "The Microarchitecture of the Pentium® 4 processor", Intel Technology Journal, 1^{st} quarter, 2001, http://developer.intel.com/technology/itj/q12001/articles/art_2.htm

[6] "UltraSPARC™ Iii User's Manual", Sun Microsystems, 1999

[7] PowerPC™ , "MPC7400 RISC Microprocessor Technical Summary ", Mororola, Order Number: MPC7400TS/D, Rev. 0, 8/1999

[8] Victor Kartunov, "IBM PowerPC G5: Another World", X-bit Labs, Jan. 2004 http://www.xbitlabs.com/articles/cpu/display/powerpc-g5_6.html

[9] N. Jouppi, "Improving direct-mapped cache performance by the addition of a small fully associative cache and prefetch buffers", International Symposium on. Computer Architecture 1990

[10] Jeffrey B. Rothman, Alan Jay Smith: "Sector Cache Design and Performance". International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems, 2000

[11] Andre. Seznec. "Decoupled sectored caches: conciliating low tag implementation cost". International Symposium on. Computer Architecture, 1994

[12] J.S.Lipty. "Structural Aspects of the System/360 Model 85, Part II: The Cache. IBM Systems Journal, Vol. 7, 1968

[13] Jeffrey B. Rothman and Alan Jay Smith. "The Pool of SubSectors Cache Design". International Conference on Supercomputing, 1999

[14] Mark D. Hill and Alan Jay Smith. "Experimental Evaluation of On-Chip Microprocessor Cache Memories". International Symposium on Computer Architecture, June 1984

[15] James R. Goodman. "Using Cache Memory to Reduce Processor Memory Traffic". International Symposium on. Computer Architecture 1983

[16] G. Albera and R. Bahar, " Power/performance Advantages of Victim Buffer in High-Performance Processors", IEEE Alessandro Volta Memorial Workshop on Low-Power Design,1999

[17] Farhad Shafai, Kenneth J. Schultz, G..F. Randall Gibson, Armin G. Bluschke and David E. Somppi, "Fully Parallel 30-MHz, 2.5-Mb CAM", IEEE journal of solid-state circuits, Vol. 33, No. 11, November 1998

[18] SPEC CPU2000, http://www.specbench.org/osg/cpu2000

[19] SPEC JBB 2000, http://www.specbench.org/jbb2000

[20] C.Lai, S. Lu and Q. Zhao, "Performance Analysis of Speech Recognition Software", Workshop on Computer Architecture Evaluation using Commercial Workloads, International Symposium on High Performance Computer Architecture, 2002

[21] J. Song, J. Li, and Y.-K. Chen, "Quality-Delay and Computation Trade-Off Analysis of Acoustic Echo Cancellation On General-Purpose CPU," International Conference on Acoustics, Speech, and Signal Processing, 2003.

[22] R. Uhlig et. al., "SoftSDV: A Pre-silicon Software Development Environment for the IA-64 Architecture", Intel Technology Journal, 4th quarter, 1999. http://developer.intel.com/technology/itj/q41999/articles/art_2.htm

Appendix: More Simulation Data

Sector/ line size		16B/ 32B	16B/ 64B	32B/ 64B	16B/ 128B	32B/ 128B	64B/ 128B	32B/ 256B	64B/ 256B	128B/ 256B
LVCSR	LVC	2.52	2.64	1.68	2.70	1.72	1.17	1.74	1.18	0.87
	SVC	2.53	2.69	1.70	2.83	1.78	1.19	1.87	1.24	0.88
	VST	2.59	2.73	1.79	2.90	1.89	1.38	2.02	1.51	1.20
	ORI	2.65	2.92	1.92	3.23	2.15	1.55	2.86	2.19	1.69
AMMP	LVC	8.17	8.58	5.32	9.00	5.64	3.88	5.74	3.96	3.04
	SVC	8.23	8.77	5.37	9.36	5.76	3.90	6.12	4.12	3.06
	VST	8.27	8.71	5.46	9.09	5.72	4.02	5.90	4.16	3.40
	ORI	8.38	9.08	5.67	9.86	6.25	4.37	6.68	4.69	3.66
MESA	LVC	0.53	0.54	0.31	0.54	0.31	0.19	0.31	0.20	0.14
	SVC	0.54	0.55	0.32	0.63	0.33	0.20	0.53	0.25	0.15
	VST	0.95	1.03	0.75	1.08	0.78	0.69	0.86	0.75	1.39
	ORI	1.36	1.67	1.20	2.72	1.98	1.62	2.72	2.29	1.93
SAEC	LVC	3.11	3.07	1.60	2.95	1.54	0.83	1.45	0.78	0.45
	SVC	3.16	3.25	1.66	3.38	1.73	0.89	1.83	0.93	0.49
	VST	3.22	3.27	1.72	3.33	1.75	0.96	1.80	0.99	0.58
	ORI	3.25	3.35	1.75	3.50	1.83	0.99	1.98	1.08	0.62
GZIP	LVC	9.09	10.0	8.28	10.7	9.00	7.54	9.36	7.95	6.57
	SVC	9.10	10.1	8.29	11.0	9.16	7.59	10.0	8.38	6.76
	VST	9.18	10.13	8.46	10.86	9.26	8.02	9.83	8.66	7.97
	ORI	9.54	10.8	9.09	12.1	10.5	9.09	12.1	10.8	9.48
GCC	LVC	1.92	1.76	0.95	1.46	0.79	0.45	0.46	0.27	0.18
	SVC	2.05	2.23	1.09	2.49	1.19	0.57	1.36	0.63	0.31
	VST	2.04	2.07	1.15	2.08	1.16	0.69	1.17	0.69	0.45
	ORI	2.14	2.35	1.25	2.64	1.40	0.77	1.60	0.88	0.51
SJBB	LVC	3.55	3.67	2.31	3.72	2.34	1.54	2.36	1.55	1.08
	SVC	3.59	3.77	2.34	3.97	2.45	1.58	2.70	1.70	1.11
	VST	3.62	3.78	2.40	3.91	2.50	1.72	2.47	1.89	1.41
	ORI	3.70	3.96	2.52	4.33	2.79	1.89	3.34	2.83	1.68

L1 Data		DM	2way	4way	8way	16 Way	DM(8 Entries)	8KB	16KB	32KB	64KB	128KB	256KB
LVCSR	LVC	2.71	2.68	2.64	2.61	2.60	3.35	2.77	2.64	2.50	2.35	2.18	1.92
	SVC	2.75	2.71	2.69	2.64	2.63	3.70	2.90	2.69	2.52	2.36	2.18	1.93
	VST	4.16	3.10	2.73	2.63	2.62	4.33	3.01	2.73	2.52	2.35	2.18	1.92
	ORI	5.31	3.64	2.92	2.69	2.66	5.31	3.35	2.92	2.58	2.37	2.19	1.93
AMMP	LVC	9.11	8.76	8.58	8.51	8.49	9.76	9.47	8.58	7.92	7.64	7.44	6.20
	SVC	9.35	9.01	8.77	8.65	8.65	9.89	9.80	8.77	8.00	7.66	7.45	6.22
	VST	10.08	8.95	8.71	8.61	8.59	10.44	9.89	8.71	7.96	7.65	7.44	6.13
	ORI	11.36	9.38	9.08	8.94	8.94	11.36	10.27	9.08	8.10	7.68	7.45	6.23
MESA	LVC	0.56	0.54	0.54	0.54	0.55	4.10	0.63	0.54	0.41	0.30	0.16	0.10
	SVC	0.74	0.55	0.55	0.56	0.57	4.91	0.67	0.55	0.44	0.31	0.17	0.10
	VST	4.08	1.56	1.03	0.60	0.58	4.94	1.48	1.03	0.45	0.35	0.18	0.10
	ORI	6.68	3.02	1.67	0.66	0.60	6.68	3.22	1.67	0.47	0.32	0.17	0.11
SAEC	LVC	3.27	3.15	3.07	3.01	3.02	4.09	3.82	3.07	2.27	1.20	1.00	0.64
	SVC	3.76	3.37	3.25	3.23	3.23	4.26	4.19	3.25	2.39	1.31	1.01	0.65
	VST	4.22	3.43	3.27	3.24	3.23	4.38	4.38	3.27	2.31	1.29	1.00	0.64
	ORI	4.72	3.56	3.35	3.32	3.32	4.72	4.59	3.35	2.43	1.35	1.01	0.65
GZIP	LVC	10.30	10.13	10.04	10.00	9.97	11.53	11.27	10.04	8.08	5.05	1.81	0.37
	SVC	10.35	10.21	10.12	10.07	10.04	11.50	11.35	10.12	8.19	5.19	1.91	0.38
	VST	14.62	10.30	10.13	10.07	10.03	15.51	11.58	10.13	8.13	5.05	1.83	0.38
	ORI	16.32	11.17	10.81	10.68	10.62	16.32	12.90	10.81	8.65	5.47	2.02	0.39
GCC	LVC	1.55	1.69	1.76	1.82	1.82	1.98	3.82	1.76	0.32	0.21	0.17	0.15
	SVC	1.88	2.17	2.23	2.24	2.19	2.00	4.46	2.23	0.37	0.22	0.17	0.15
	VST	1.86	2.00	2.07	2.19	2.23	2.05	4.32	2.07	0.36	0.22	0.18	0.15
	ORI	2.30	2.26	2.35	2.43	2.47	2.30	4.47	2.35	0.49	0.23	0.19	0.15
SJBB	LVC	3.76	3.71	3.67	3.67	3.68	4.38	4.05	3.67	3.19	2.58	2.00	1.76
	SVC	4.07	3.89	3.77	3.75	3.75	4.56	4.42	3.77	3.26	2.63	2.03	1.76
	VST	4.56	3.98	3.78	3.73	3.71	4.86	4.42	3.78	3.25	2.63	2.06	1.78
	ORI	5.37	4.23	3.96	3.84	3.80	5.37	5.00	3.96	3.32	2.66	2.04	1.77

L2 Data		256KB	512KB	1MB	2MB	4MB	8MB	DM	2way	4way	8way	16way
Ammp	LVC	55.76	38.38	13.71	8.21	7.36	2.28	35.49	33.31	19.93	13.71	12.59
	SVC	56.64	39.05	14.58	8.27	7.41	2.31	36.26	34.04	20.67	14.58	13.00
	VST	54.37	36.69	13.30	8.24	7.39	2.27	36.13	33.16	18.76	13.30	12.59
	ORI	57.57	40.07	15.62	8.49	7.58	2.34	38.18	35.11	21.65	15.62	13.75
LVCS	LVC	52.63	46.03	36.33	26.49	20.72	14.66	40.69	37.79	36.65	36.33	36.08
	SVC	52.89	46.21	36.49	26.57	20.75	14.68	41.31	38.00	36.83	36.49	36.23
	VST	52.81	46.11	36.36	26.51	20.72	14.67	41.32	37.88	36.70	36.36	36.10
	ORI	53.28	46.40	36.60	26.63	20.76	14.69	42.29	38.19	36.96	36.60	36.35
SAEC	LVC	14.86	0.54	0.37	0.36	0.34	0.32	7.16	0.52	0.38	0.37	0.36
	SVC	15.63	0.82	0.38	0.36	0.34	0.32	7.64	0.64	0.45	0.38	0.36
	VST	14.70	0.87	0.42	0.36	0.34	0.32	7.45	0.76	0.45	0.42	0.36
	ORI	15.78	1.07	0.41	0.36	0.34	0.32	8.48	0.95	0.49	0.41	0.36

Cache Behavior Analysis of a Compiler-Assisted Cache Replacement Policy*

Xingyan Tian[1], Kejia Zhao[1], Huowang Chen[1], and Hongyan Du[2]

[1] Department of Computer Science, National University of Defense Technology,
Changsha, Hunan, 410073, China
tianxingyan@nudt.edu.cn

[2] Department of Computer Science, Changsha University,
Changsha, Hunan, 410003, China
csduhongyan@sina.com

Abstract. Recent research results show that conventional hardware-only cache replacement policies result in unsatisfactory cache utilization because of cache pollution. To overcome this problem, cache hints are introduced to assist cache replacement. Cache hints are used to specify the cache level at which the data is stored after accessing it. This paper present a compiler-assisted cache replacement policy, Optimum Cache Partition (OCP), which can be carried out through cache hints and LRU replacement policy. Presburger arithmetic is used to exactly model the behavior of loop nests under OCP policy. The OCP replacement policy results in plain cache behaviors, and makes cache misses analyzing and optimizing easily and efficiently. OCP replacement policy has been implemented in our compiler test-bed and evaluated on a set of scientific computing benchmarks. Initial results show that our approach is effective on reducing the cache miss rate.

1 Introduction

Caches play a very important role in the performance of modern computer systems due to the gap between the memory and the processor speed, but they are only effective when programs exhibit data locality. In the past, many compiler optimizations have been proposed to enhance the data locality. However, a conventional cache is typically designed in a hardware-only fashion, where data management including cache line replacement is decided purely by hardware. Research results [1] reveal that considerable fraction of cache lines are held by data that will not be reused again before it is displaced from the cache. This phenomenon, called cache pollution, severely degrades cache performance. A consequence of this design approach is that cache can make poor decisions in choosing data to be replaced, which may lead to poor cache performance.

* This work was supported by 863 National High Technology Program, grant No. 2002AA1Z2105

P.-C. Yew and J. Xue (Eds.): ACSAC 2004, LNCS 3189, pp. 30–43, 2004.

There are a number of efforts in architecture designed to address this problem, the cache hint in EPIC [2,3] (Explicitly Parallel Instruction Computing) architectures is one of them. Cache hints are used to specify the cache level where the data is stored after accessing it. Intuitively, two kinds of memory instructions should be given cache hints [12]: i) whose referenced data doesn't exhibit reuse; ii) whose referenced data does exhibit reuse, but it cannot be realized under the particular cache configuration. It sounds as though the problem is pretty simple for regular applications, and existing techniques for analyzing data reuse [4] and estimating cache misses [5, 6] suffice to solve this problem. This plausible statement, however, is not true because a fundamental technique used in cache miss estimation — footprint analysis — is based on the assumption that all accessed data compete for cache space equally. However, in EPIC architectures, memory instructions are not homogeneous— those with cache hints have much less demand for cache space. This makes the approach derived from traditional footprint analysis very conservative. In summary, the following *cyclic dependence* [12] exists: Accurate cache miss estimation must be known to cache hint assignment, while accurate cache miss estimation is only possible when cache hint assignment is finalized.

In this paper, we present a novel cache replacement policy, Optimum Cache Partition (OCP), to address the above problem. The OCP cache replacement policy can be carried out through combining hardware cache replacement policy LRU (Least Recently Used) with compiler generating cache hints. Presburger [15] arithmetic is used to exactly model the behavior of loop nests under OCP policy. the OCP replacement policy makes cache miss estimation simpler through simplifying cache behaviors fundamentally.

We have evaluated the benefit of the OCP cache replacement policy on reducing the cache miss rate through executing a set of SPEC benchmarks on Trimaran[7] and DineroIV[8]. Trimaran is a compiler infrastructure for supporting state of the art research in EPIC architectures, and DineroIV is a trace-driven cache simulator. Initial experimental results show that our approach reduces data cache misses by 20.32%.

The rest of this paper is organized as follows. Section 2 briefly reviews the basic concept of cache hint. Section 3 illustrates, through an example, a compiler-assisted cache replacement policy, Cache Partition. Section 4 uses Presburger arithmetic to analyze the relationship between reuse vector and cache miss rate, to estimate cache hits and misses, and then an optimum cache replacement policy, Optimum Cache Partition, is derived. Our implementation and experimental results are then presented in Section 5. Section 6 discusses related work. Section 7 concludes this paper.

2 Cache Hints

One of the basic principles of the EPIC-philosophy is to let the compiler decide when to issue operations and which resources to use. The existing EPIC architectures (HPL-PD [2] and IA-64[3]) communicate the compiler decisions about the cache hierarchy management to the processor through cache hints. The semantics of cache hints on both architectures are similar. Cache hints are used to specify the cache level at which the data is likely to be found, as well as the cache level where the data is stored after accessing it.

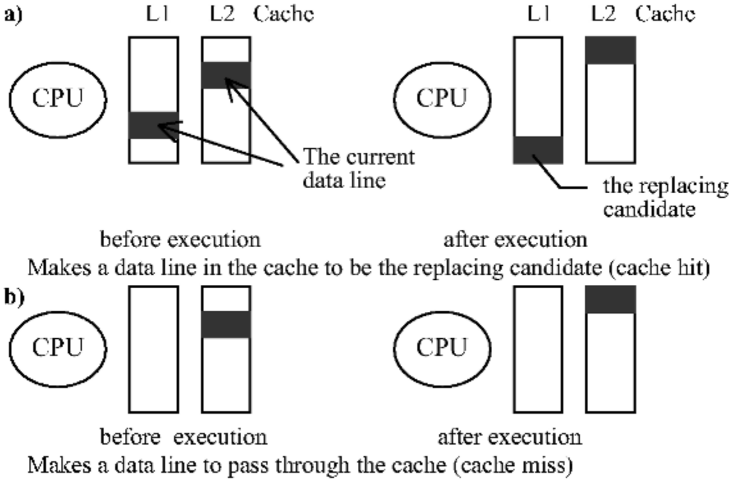


Fig. 1. Example of the effect of the cache hints in the load instruction *ld.nt1*

In the IA-64 architecture, the cache hints *t1*, *nt1*, *nt2* and *nta* are defined. *t1* means that the memory instruction has temporal locality in all the cache levels. *nt1* only has temporal locality in the L2 cache and below. Similarly, *nt2* respectively *nta* indicates that there's only temporal locality in L3 respectively no temporal locality at all. An example is given in figure 1 where the effect of the load instruction *ld.nt1* is shown.

Generating cache hints based on the data locality of the instruction, compiler can improve a program's cache behavior. As shown in figure 1, cache hints have two effects on cache behaviors: i) when cache hits, it make the current reference line, which is stayed in the cache, to be the replacing candidate, ii) when cache misses, it make the current reference line, which is not in the cache, to pass through the cache without being stored in the cache. Examples of these effects are given in figure 1a and 1b.

With cache hints, our compiler can carry out the OCP cache replacement policy through controlling cache staying time of accessed data, as a result, the cache pollution is reduced and cache miss rate is decreased availably.

3 A Compiler-Assisted Cache Replacement Policy: Cache Partition

Let's see a short memory access stream in Figure 2. For a 4-lines LRU cache, there are 10 cache misses for the memory access stream in Figure 2, M_{LRU}=10.

For the same cache, if the cache is partitioned into three parts logically: 3 lines, 0 lines, 1 line, and then the three parts has been assigned to the three references, A, B and C respectively. In that case, memory access stream of the reference A can hold 3 cache lines at most, the reference C can only hold 1 cache line, the reference B can not hold any cache line. Under this cache partition, the cache miss analysis of these 3 references is shown in Figure 3. It shows that the cache miss analysis of these 3 references can proceed independently, there are no influence mutually.

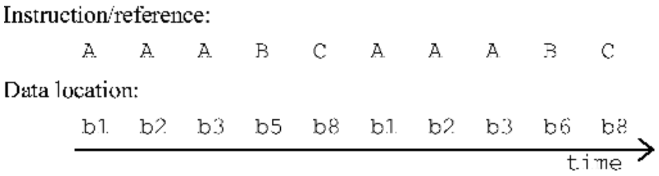


Fig. 2. The top row indicates 10 memory reference instructions. The bottom row shows the corresponding memory locations.

Making reasonable cache partition and limiting the appropriate number of cache lines hold by each reference, cache misses can decrease obviously. The "limiting" can be carried out by a compiler through appending cache hints to some memory instructions. For example, not to let reference B hold any cache line, a cache hint is appended to reference B instruction; To limit the number of cache lines hold by references A and C, a cache hint is appended to reference A and C instruction conditionally.

Cache lines	4		
Cache Partition	3	0	1
Reference	A	B	C
Access stream	b1,b2,b3,b1,b2,b3,	b5,b6	b8,b8
Cache misses	3	2	1

Cache miss analysis for Cache Partition (3,0,1)

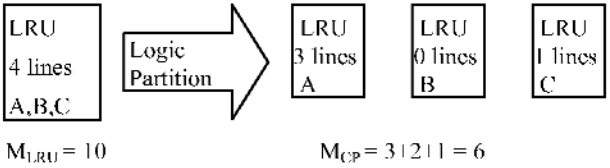


Fig. 3. 4-lines LRU cache is divided logically into three LRU sub-caches, and cache misses decrease from 10 to 6.

With cache hints, as above example shows, a compiler can construct a novel cache replacement policy, where the cache has be partitioned logically into parts and each reference data in a loop nested program can be limited within one part, as if there is a *sub-cache* for each reference. Under this cache replacement policy, reference cache behaviors will not influence each other, and cache misses of a reference can be analyzed independently. The cache behaviors under the cache replacement policy are more plain and more independent than only-LRU replacement policy.

Definition 1. For a LRU cache with *N* lines, a *Cache Partition* $\mathcal{P}$ *of* a loop nest with *m* references is a m-tuple (C_1, C_2, ..., C_m), where $\sum_{i=1}^{m} C_i \le N$ and cache lines hold by *i*-th reference Ref_i at any moment can not exceeds C_i lines.

A Cache Partition (CP) is a cache replacement policy that achieves two aims for the reference Ref_i: (i) there are C_i lines in cache that can be hold by the reference Ref_i at any moment; (ii) the number of cache lines hold by the reference Ref_i cannot exceed C_i at any moment. The CP replacement policy, that can be carried out by a compiler through appending cache hints to some memory instructions, simplifies loop nest cache behaviors, however there is a crucial problem: can it minify cache misses? To address this problem, an analysis of cache misses under the CP policy is taken, and an optimum CP policy is derived in the next section.

4 Cache Misses Analysis and Optimum Cache Partition

In this section, the Optimum Cache Partition is derived after cache misses analysis under the CP replacement policy. A matrix-multiply loop nest program MXM(see Figure 4) is used as our primary example, and all arrays discussed here are assumed to be arranged in column-major order as in Fortran.

```
DO i = 1, U1
DO k = 1, U2
DO j = 1, U3
   Z(j,i)=Z(j,i)+X(k,i)*Y(j,k)
```

Fig. 4. Matrix-multiply loop nest, MXM

4.1 Terminology

Our research is based on analyzing references' cache behavior using iteration spaces and reuse vectors.

A *reference* is a static read or write in the program, while a particular execution of that read or write at runtime is a *memory access*.

Throughout this paper, we denote the cache size as C_s, line size as L_s, the number of cache lines as N_l, then $C_s = L_s * N_l$.

Iteration space. Every iteration of a loop nest is viewed as a single entity termed *an iteration point* in the set of all iteration points known as the *iteration space*. Formally, we represent a loop nest of depth *n* as a finite convex polyhedron of the *n*-dimensional iteration space $S = \prod_{i=1}^{n}[1, U_i] \subset Z^n$, bounded by the loop bounds U_i ($1 \le i \le n$). Each iteration in the loop corresponds to a node in the polyhedron and is called an *iteration point*. Every iteration point is identified by its index vector $\vec{i} = (i_1, i_2, ..., i_n) \in S$, where i_l is the loop index of the l-th loop in the nest with the outermost loop corresponding to the leftmost index. In this representation, if iteration $\vec{p}_2$ executes after iteration $\vec{p}_1$ we write $\vec{p}_2 \succ \vec{p}_1$ and say that $\vec{p}_2$ is lexicographically greater than $\vec{p}_1$.

Reuse Vector. Reuse vectors provide a mechanism for summarizing repeated memory access patterns in loop-oriented code[4]. If a reference accesses the same memory line in iterations $\vec{i}_1$ and $\vec{i}_2$, where $\vec{i}_2 \succ \vec{i}_1$, we say that there is reuse in direction $\vec{r} = \vec{i}_2 - \vec{i}_1$, and $\vec{r}$ is called a *reuse vector.* For example, the reference Z(j, i) in Figure 4 can access the same memory line at the iteration points (i, k, j) and (i, k+1, j), and hence one of its reuse vectors is (0, 1, 0). A reuse vector is repeated across the iteration space. Reuse vectors provide a concise mathematical representation of the reuse information of a loop nest.

A reuse vector is realized. If a reuse vector results in a cache hit we say that the *reuse vector is realized.*

Hence, if we have an infinitely large cache, every reuse vector would result in a cache hit. In practice, however, a reuse vector does not always result in cache hit. The central idea behind our model is to maximize realized reuse vectors.

4.2 Cache Misses Analysis Under Cache Partition Replacement Policy

Under a Cache Partition $\mathcal{P}= (C_1, C_2,\ldots, C_m)$ replacement policy, memory accesses of every reference Ref_i $(1 \leq i \leq m)$ in a loop nest can hold C_i cache lines at most, and cannot be influenced by other reference accesses, as if reference Ref_i engrosses a C_i-lines sub-cache, $Cache_i$, independently (see Figure 5). So, the cache behavior under Cache Partition $\mathcal{P} = (C_1, C_2,\ldots, C_m)$ replacement policy is same as the cache behavior of every reference Ref_i $(1 \leq i \leq m)$ in $Cache_i$ with C_i cache lines respectively.

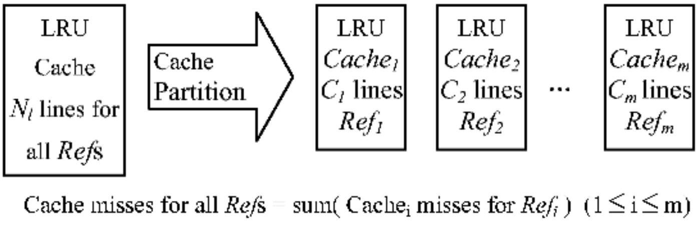


Fig. 5. Cache Partition Replacement Policy $\mathcal{P}= (C_1, C_2,\ldots, C_m)$

Now, we use Presburger arithmetic[15] to exactly model the cache behavior of reference Ref_i $(1 \leq i \leq m)$ in $Cache_i$ with C_i cache lines. Mostly cache hits of a reference in a loop nest are produced by one or two reuse vectors of the reference, so we analyze primarily the cache behavior of a reference while only one or two reuse vectors of the reference are realized, but these conditions, that more than two reuse vectors are realized, are considered also. The cache behavior analyses have four steps:

1) When a reuse vector of a reference is realized in whole iteration space, the cache-hit iterations of the reference are formulated. For reuse vectors $\vec{r}$ of the reference *Ref*$_i$, a formula CHI($\vec{r}$) is generated. CHI($\vec{r}$) represents all cache-hit iterations when reuse vector $\vec{r}$ is realized in whole iteration space;
2) The number of cache lines, that *cache*$_i$ needed for to realize a reuse vector $\vec{r}$ of *Ref*$_i$ in whole iteration space, is counted;
3) After steps 1 and 2, a *hits-cost pair, <Hits, Cost>*, for every reuse vector $\vec{r}$ of the reference *Ref*$_i$ is constructed. This *hits-cost pair <Hits, Cost>* means that: if cache lines assigned to *Ref*$_i$ is not less than the *cost*, namely $Cost \leq C_i$, then *Ref*$_i$ would produce cache hit *hits* times in whole iteration space by it's reuse vector $\vec{r}$. The set of hits-cost pairs of *Ref*$_i$ is noted as HCSet$_i$.
4) According to HCSet$_i$ ($1 \leq i \leq m$), $H_{\mathcal{P}}$, the number of cache hits under the Cache Partition $\mathcal{P}$ replacement policy, is estimated, and then the Optimum Cache Partition, the Cache Partition that results in maximum cache hits, is derived.

The four steps are explained in further detail below.

1. Cache-Hit Iterations

When a reuse vector $\vec{r}$ of reference *Ref*$_i$ is realized in whole iteration space *S*, the set of *cache-hit iterations* of reference *Ref*$_i$ is formulated as CHI($\vec{r}$):

$$\text{CHI}(\vec{r}) = \{\vec{p} \mid (\vec{p} \in S) \wedge (\exists \vec{p}': (\vec{p}' \in S) \wedge (\vec{p} = \vec{p}' + \vec{r}))\} \quad (1)$$

The above formula means that reference *Ref*$_i$ at iteration $\vec{p}$ can reuse through reuse vector $\vec{r}$ if and only if there is another iteration $\vec{p}'$ that can be hit by $\vec{p}$ through reuse vector $\vec{r}$, expressed as $\vec{p} = \vec{p}' + \vec{r}$.

The number of cache-hit iterations in CHI($\vec{r}$) can be calculated as following: (consider $\vec{r} = (r_1, r_2, \ldots, r_n)$)

$$|\text{CHI}(\vec{r})| = \prod_{i=1}^{n} (U_i - |r_i|) \quad (1.1)$$

For a spatial reuse vector $\vec{r}_s = (r_{s1}, r_{s2}, \ldots, r_{sn})$, formulas (1) and (1.1) is unsuitable. Cache behaviors of a loop nest with a spatial reuse vector are complicated. To get a concise formula for a spatial reuse vector, we modified formula (1.1) by appending a coefficient without a complicated model:

$$|\text{CHI}(\vec{r}_s)| = (\prod_{i=1}^{n} (U_i - |r_{si}|)) \times (\frac{\mathrm{b}\vec{\mathrm{r}}_{\mathrm{s}} - 1}{\mathrm{b}\vec{\mathrm{r}}_{\mathrm{s}}}) \quad (1.2)$$

In formula (1.2), b$\vec{r}_s$ stands for iteration times of the reference *Ref*$_i$ in a single cache line along the spatial reuse vector $\vec{r}_s$. Among the b$\vec{r}_s$ times iterations in a single cache line, there are (b$\vec{r}_s$-1) iterations to be reused, so the cache hits number of a spatial reuse vector has a coefficient (b$\vec{r}_s$-1)/b$\vec{r}_s$.

When two reuse vectors $\vec{r}_1$ and $\vec{r}_2$ of reference Ref_i are realized in whole iteration space S, the set of *cache-hit iterations* of reference Ref_i is formulated as CHI($\vec{r}_1, \vec{r}_2$):

$$\mathrm{CHI}(\vec{r}_1, \vec{r}_2) = \{\vec{p} \mid (\vec{p} \in S) \wedge (\exists \vec{p}' : (\vec{p}' \in S) \wedge (\vec{p} = \vec{p}' + \vec{r}_1 \vee \vec{p} = \vec{p}' + \vec{r}_2))\} \quad (1')$$

When k reuse vectors $\vec{r}_1, \vec{r}_2, \ldots,$ $\vec{r}_k$ of reference Ref_i are realized in whole iteration space S, the set of *cache-hit iterations* of reference Ref_i is formulated as CHI($\vec{r}_1, \vec{r}_2,$ …, $\vec{r}_k$):

$$\mathrm{CHI}(\vec{r}_1, \vec{r}_2, \ldots, \vec{r}_k) =$$
$$\{\vec{p} \mid (\vec{p} \in S) \wedge (\exists \vec{p}' : (\vec{p}' \in S) \wedge (\vec{p} = \vec{p}' + \vec{r}_1 \vee \vec{p} = \vec{p}' + \vec{r}_2 \vee \ldots \vee \vec{p} = \vec{p}' + \vec{r}_k))\} \quad (1'')$$

2. The Number of Cache Lines

This step is to count the number of cache lines NCL($\vec{r}$), that $cache_i$ needed for to realize a reuse vector $\vec{r}$ of Ref_i in whole iteration space. First, the iteration set ACI($\vec{r}, \vec{q}$) is defined. At any iteration $\vec{q} \in S$, if these data lines, accessed by Ref_i at these iterations in ACI($\vec{r}, \vec{q}$), is hold by $cache_i$, then the reuse vector $\vec{r}$ would be realized in whole iteration space. Furthermore, the number of cache lines NCL($\vec{r}$) in $cache_i$ is derived.

$$\forall \vec{q} \in S : \mathrm{ACI}(\vec{r}, \vec{q}) = \{\vec{p} \mid (\vec{p} \in S) \wedge (\vec{p} + \vec{r} \in S) \wedge (\vec{p} \prec \vec{q}) \wedge (\vec{p} + \vec{r} \succ \vec{q})\} \quad (2)$$

ACI($\vec{r}, \vec{q}$) is similar as reference windows[14] which hold by a cache can result in the reuse vector $\vec{r}$ realized.

$$\mathrm{NCL}(\vec{r}) = \max\{ |\mathrm{ACI}(\vec{r}, \vec{q})| \mid \vec{q} \in S \} \quad (3)$$

When cache lines in $cache_i$ is not less than NCL($\vec{r}$), $cache_i$ can hold all data lines accessed at these iterations in ACI($\vec{r}, \vec{q}$), and then the reuse vector $\vec{r}$ would be realized in whole iteration space.

To avoid plentiful calculation for ACI($\vec{r}, \vec{q}$), we give a simple estimation formula (3.1). We observe that |ACI($\vec{r}$, $\vec{q}$)| is not greater than $|\{\vec{p} \mid (\vec{p} \in S) \wedge (\vec{p} \prec \vec{q}) \wedge (\vec{p} + \vec{r} \succ \vec{q})\}|$ that is the number of the iteration points within a reuse vector $\vec{r}$ in the loop nest iteration space, so NCL($\vec{r}$) can be estimated as:

$$\mathrm{NCL}(\vec{r}) \cong \sum_{i=1}^{n} (r_i * \prod_{j=i+1}^{n} U_j) \text{ where } \prod_{j=n+1}^{n} U_j = 1 \quad (3.1)$$

When two reuse vectors $\vec{r}_1$ and $\vec{r}_2$ of reference Ref_i are realized in whole iteration space S, the ACI($\vec{r}_1, \vec{r}_2, \vec{q}$) and NCL($\vec{r}_1, \vec{r}_2$) is formulated as following: (consider $\vec{r}_1 \prec \vec{r}_2$)

$$\forall \vec{q} \in S : \mathrm{ACI}(\vec{r}_1, \vec{r}_2, \vec{q}) = \mathrm{ACI}(\vec{r}_1, \vec{q}) \cup$$
$$\{\vec{p} \mid (\vec{p} \in S) \wedge (\vec{p} + \vec{r}_2 \in S) \wedge (\vec{p} \prec \vec{q}) \wedge (\vec{p} + \vec{r}_2 \succ \vec{q}) \wedge (\vec{p} + \vec{r}_2 - \vec{r}_1 \notin S)\} \quad (2')$$

$$\mathrm{NCL}(\vec{r}_1, \vec{r}_2) = \max\{\ |\mathrm{ACI}(\vec{r}_1, \vec{r}_2, \vec{q})| \ \big|\ \vec{q} \in S\ \} \quad (3')$$

When two reuse vectors $\vec{r}_1, \vec{r}_2, \ldots, \vec{r}_k$ of reference Ref_i are realized in whole iteration space S, the ACI($\vec{r}_1, \vec{r}_2, \ldots, \vec{r}_k, \vec{q}$) and NCL($\vec{r}_1, \vec{r}_2, \ldots, \vec{r}_k$) is formulated as following: (consider $\vec{r}_1 \prec \vec{r}_2 \prec \ldots \prec \vec{r}_k$)

$$\forall \vec{q} \in S : \mathrm{ACI}(\vec{r}_1, \vec{r}_2, \ldots, \vec{r}_k, \vec{q}) = \mathrm{ACI}(\vec{r}_1, \vec{r}_2, \ldots, \vec{r}_{k-1}, \vec{q}) \cup$$
$$\{\vec{p} \mid (\vec{p} \in S) \wedge (\vec{p} + \vec{r}_k \in S) \wedge (\vec{p} \prec \vec{q}) \wedge (\vec{p} + \vec{r}_k \succ \vec{q}) \wedge$$
$$(\vec{p} + \vec{r}_k - \vec{r}_1 \notin S) \wedge \ldots \wedge (\vec{p} + \vec{r}_k - \vec{r}_{k-1} \notin S)\} \quad (2'')$$

$$\mathrm{NCL}(\vec{r}_1, \vec{r}_2, \ldots, \vec{r}_k) = \max\{\ |\mathrm{ACI}(\vec{r}_1, \vec{r}_2, \ldots, \vec{r}_k, \vec{q})| \ \big|\ \vec{q} \in S\ \} \quad (3'')$$

3. Hits-Cost Pairs Set

This *hits-cost pair, <Hits, Cost>*, means that: if cache lines C_i, assigned to Ref_i, is not less than the *Cost*, namely $Cost \le C_i$, then Ref_i would produce cache hits *Hits* times in whole iteration space by it's reuse vectors. The set of hits-cost pairs of Ref_i is noted as HCSet$_i$.

$$\mathrm{HCSet}_i = \{\ <0, 0>, <|\mathrm{CHI}(\vec{r})|, \mathrm{NCL}(\vec{r})>, <|\mathrm{CHI}(\vec{r}_1, \vec{r}_2)|, \mathrm{NCL}(\vec{r}_1, \vec{r}_2)>, \ldots,$$
$$<|\mathrm{CHI}(\vec{r}_1, \vec{r}_2, \ldots, \vec{r}_k)|, \mathrm{NCL}(\vec{r}_1, \vec{r}_2, \ldots, \vec{r}_k)> \big|\ \vec{r}, \vec{r}_1, \vec{r}_2, \ldots, \vec{r}_k$$
$$\text{are reuse vectors of } Ref_i\ \} \quad (4)$$

The hits-cost pair <0, 0> means that: no cache line assigned to Ref_i, no cache hit produced. HCSet$_i$ describes the relations of Ref_i between the number of cache lines and the number of cache hits. To calculate HCSet$_i$ in our experiments, these hits-cost pairs, $<|\mathrm{CHI}(\vec{r}_1, \vec{r}_2, \ldots, \vec{r}_k)|, \mathrm{NCL}(\vec{r}_1, \vec{r}_2, \ldots, \vec{r}_k)>$ ($k > 2$), were ignored, there are two reasons: i) the cache lines is not enough to realize more reuse vectors; ii) most cache hits of a reference is produced by one or two reuse vectors of the reference.

4. Optimum Cache Partition

To assign C_i cache lines to reference Ref_i under the Cache Partition $\mathcal{P}$ replacement policy, the number of cache hits of the reference Ref_i can be estimated as following:

$$\mathrm{Hit}_i(C_i) = \max\ \{\ Hits\ \big|\ (<Hits, Cost> \in \mathrm{HCSet}_i) \wedge (Cost \le C_i)\ \} \quad (5)$$

Under the Cache Partition $\mathcal{P} = (C_1, C_2, \ldots, C_m)$ replacement policy, cache hits of a loop nest can be estimated as follow:

$$H_{\mathcal{P}} = \sum_{i=1}^{m} \left[Hit_i(C_i)\right] \quad (6)$$

Formula (6) estimates cache hits number of a loop nest program under a Cache Partition. Different Cache Partitions can bring different cache hits number, a Cache

Partition, that brings the most cache hits, is called Optimum Cache Partition replacement policy.

Definition 2. Optimum Cache Partition of a loop nest. $\mathcal{P}$ is a Cache Partition of a loop nest, for any other Cache Partition $\mathcal{P}'$ if this inequation $H_{\mathcal{P}} \leq H_{\mathcal{P}'}$ is always true, then $\mathcal{P}$ is called *Optimum Cache Partition (OCP) for a loop nest.*

Figure 6 shows all hits-cost pairs for a MXM program while one temporal reuse vector or one spatial reuse vector is realized. The reference *X(k, i)* has a temporal reuse vector $\vec{r}_t$ = (0,0,1), a spatial reuse vector $\vec{r}_s$ = (0,1,0). According to Formulas (1 ~ 6), when a temporal reuse vector (0,0,1) is realized, there is a Hits-Cost pair < 990000, 1> that means if there is only 1 cache line assigned to reference *X(k, i)*, *X(k, i)* would produce cache hit 990000 times by temporal reuse vector (0,0,1); when a spatial reuse vector (0,1,0) is realized, there is a Hits-Cost pair <742500, 100> that means if there is 100 cache line assigned to reference *X(k, i)*, *X(k, i)* would produce cache hit 742500 times by spatial reuse vector (0,1,0).

If there is 132 cache lines in a cache, N_l = 132, then Cache Partition $\mathcal{P}$ = (1, 1, 100) is a Optimum Cache Partition of the MXM loop nest, and under the OCP cache replacement policy the number of cache hits is estimated as following:

$H_{\mathcal{P}}$ = 990000+742500+990000 = 2722500.

Ref$_i$	$\vec{r}_t$	*<Hits, Cost>*	$\vec{r}_s$	*< Hits, Cost >*
X(k, i)	(0,0,1)	<990000, 1>	(0,1,0)	<742500,100>
Y(j, k)	(1,0,0)	<990000,10000>	(0,0,1)	<742500,1>
Z(j, i)	(0,1,0)	<990000,100>	(0,0,1)	<742500,1>

Fig. 6. Hits-cost pairs for a MXM program where U_1=U_2=U_3= 100, cache line size *Ls* = 16 bytes, size of element in arrays is 4 bytes, then b$\vec{r}_s$ = 4. If there is 132 cache lines in a cache, then Cache Partition $\mathcal{P}$ = (1, 1, 100) is a Optimum Cache Partition.

The Optimum Cache Partition(OCP) of a loop nest is a compiler-assisted cache replacement policy, that simplifies loop nest cache behavior, facilitates cache hits estimation, and decreases cache misses efficiently.

The OCP policy is carried out by the compiler according to following three primary steps: 1) Getting reuse vectors for loop nests. Reuse vectors can be calculated automatically according to some research works on loop optimization; 2) The Optimal Cache Partition is calculated automatically by the compiler; 3) Appending cache hints conditionally to realize the OCP policy.

Appending cache hints to a reference *Ref$_i$* to limit the number of cache lines hold by the reference is a difficult work if the limited cache lines number C_i is chosen discretionarily. But under the Optimum Cache Partition replacement policy, the limited cache lines number for a reference, C_i, is calculated based on it's reuse vector, and the condition for appending cache hints is straightforward.

Under OCP cache placement policy, if a reference Ref_i is limited to hold C_i cache lines, and is realized it's reuse vector $\vec{r}$, then at any loop iteration point $\vec{q}$, the condition of appending a cache hint to a reference Ref_i is that the reference Ref_i cannot produce a new reuse along the reuse vector $\vec{r}$ at the iteration point $\vec{q}$. Formally, when the point ($\vec{q}+\vec{r}$) is out of the loop iteration space, the reference Ref_i would be appended a cache hint on iteration point $\vec{q}$.

5 Experimental Results

5.1 Experimental Platform

We have implemented OCP cache replacement policy in the Trimaran compiler and evaluated its performance by running some SPEC benchmarks. Trimaran is a compiler infrastructure for supporting state of the art research in compiling for EPIC architectures. We have re-engineered the back-end of Trimaran. Implement of OCP cache replacement policy needed to update nothing but appending cache hints to some load/store instructions conditionally.

In the Trimaran compiler infrastructure, cache behavior is simulated in DineroIV that is a trace-driven cache simulator. We have extended DineroIV to support cache hints in memory instructions.

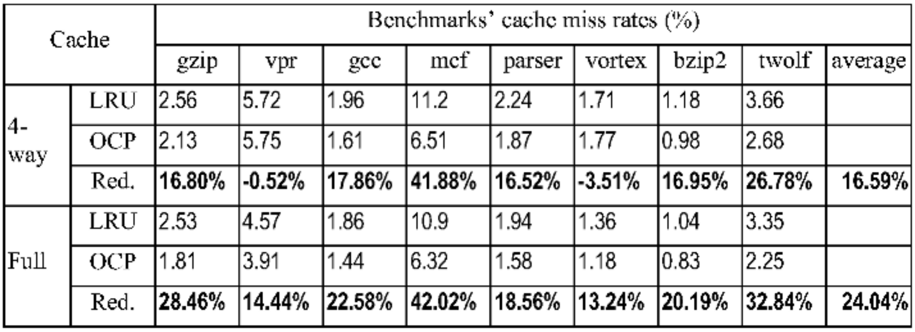
For a set associative cache, conflict misses may occur when too many data items map to the same set of cache locations. To eliminate or reduce conflict misses, we have implemented two data-layout transformations, inter- and intra-array padding[9].

5.2 Performance Results

We chosen 8 benchmarks from SPECint2000, and implemented OCP cache replacement policy on their loop kernels. We experimented our approach on 4K bytes caches with 64 bytes line size and varying associativity(4-way and full). Cache miss rates under LRU cache replacement policy and that under OCP cache replacement policy are compared in Table 1.

For a full associative cache, OCP policy is quite effective for all chosen benchmarks with average 24.04% cache misses reduction. For 4-way associative cache, OCP policy, reducing the number of cache misses by 16.59% averagely, is also quite effective except *vpr* and *vortex* benchmarks. The percentage reduction achieved on a 4-way cache is lower than that achieved by a full associative cache. This could be due to conflict misses produced by a set associative cache. The *vpr* and *vortex* benchmarks were likely to produce more conflict misses under OCP policy than under LRU replacement policy, so their cache miss rates under OCP is greater than under LRU in Table 1.

Table 1. Effective of OCP cache replacement policy in reducing cache misses. Row "LRU" reports cache miss rates under LRU replacement policy. Row "OCP" reports cache miss rates under OCP replacement policy. Row "Red." Gives the percentage reduction in cache miss rates due to our approach. There is average 20.32% cache misses reduction.



Cache		Benchmarks' cache miss rates (%)								
		gzip	vpr	gcc	mcf	parser	vortex	bzip2	twolf	average
4-way	LRU	2.56	5.72	1.96	11.2	2.24	1.71	1.18	3.66	
	OCP	2.13	5.75	1.61	6.51	1.87	1.77	0.98	2.68	
	Red.	**16.80%**	**-0.52%**	**17.86%**	**41.88%**	**16.52%**	**-3.51%**	**16.95%**	**26.78%**	**16.59%**
Full	LRU	2.53	4.57	1.86	10.9	1.94	1.36	1.04	3.35	
	OCP	1.81	3.91	1.44	6.32	1.58	1.18	0.83	2.25	
	Red.	**28.46%**	**14.44%**	**22.58%**	**42.02%**	**18.56%**	**13.24%**	**20.19%**	**32.84%**	**24.04%**

6 Related Work

Improving cache performance by cache hints has attracted a lot of attention from both the architecture and compiler perspective. In [10], keep and kill instructions are proposed by Jain et al. The keep instruction locks data into the cache, while the kill instruction indicates it as the first candidate to be replaced. Jain et al. also proof under which conditions the keep and kill instructions improve the cache hits rate. In [11], it is proposed to extend each cache line with an EM(Evict Me)-bit. The bit is set by software, based on a locality analysis. If the bit is set, that cache line is the first candidate to be evicted from the cache. These approaches all suggest interesting modifications to the cache hardware, which allow the compiler to improve the cache replacement policy.

Kristof Beyls et al proposed a framework to generate both source and target cache hints from the reuse distance metric in this paper [13]. Since the reuse distance indicates cache behavior irrespective of the cache size and associativity, it can be used to make caching decisions for all levels of cache simultaneously. Two methods were proposed to determine the reuse distances in the program, one based on profiling which statically assigns a cache hint to a memory instruction and one based on analytical calculation which allows to dynamically select the most appropriate hint. The advantage of the profiling-based method is that it works for all programs. The analytical calculation of reuse distances is applicable to loop-oriented code and has the advantage that the reuse distance is calculated independent of program input and for every single memory access. But their work, generating cache hints from the reuse distance, does have the cyclic dependency problem [12] mentioned in Section 1: Accurate reuse distance estimation must be known to cache hint assignment, while accurate reuse distance estimation is only possible when cache hint assignment is finalized. They ignored the impact of cache hints on the reuse distance.

Hongbo Yang et al [12] studied the relationship between cache miss rate and cache-residency of reference windows, known that in order for an array reference to realize its temporal reuse, its reference window must be fully accommodated in the cache. And then they formulated the problem as a 0/1 knapsack problem. The relationship between cache miss rate and cache-residency of reference windows is similar to the one considered in section 4.2, however our work has two major different aspects from their work:(i) we achieved cache hits analysis under the OCP cache replacement policy but they didn't in their 0/1 knapsack problem; (ii) we considered the impact of temporal and spatial reuse vectors on cache miss rate, while they have only considered the impact of reference windows that are decided by temporal reuse vectors.

7 Conclusions

EPIC architectures provide cache hints to allow the compiler to have more control on the data cache behavior. In this paper we constructed a compiler-assisted cache replacement policy, Optimum Cache Partition, which utilizes the cache more efficiently to achieve better performance. In particular, we presented a novel cache replacement policy, Cache Partition, which could be carried out through cache hints. Under the Cache Partition policy, we studied the relationship between cache hits rate and reuse vectors of a reference, and constructed hits-cost pairs of the reference. A hits-cost pair described a case: how many cache lines assigned to a reference could produce how many cache hits. After formulating cache hits number of a loop nest program under a Cache Partition, we could achieve Optimum Cache Partition replacement policy. To the best of our knowledge, the OCP cache replacement policy is the simplest effective cache optimization with cache hints, that results in plain cache behaviors and makes cache misses analyzing and optimizing easily and efficiently.

We evaluated our OCP cache replacement policy by implementing it in the Trimaran compiler and simulating cache behaviors in DineroIV. Our simulation results show that OCP policy exploited the architecture potential well. It reduced the number of data cache misses by 20.32% averagely.

References

1. Kathryn S. McKinley and Olivier Temam.: Quantifying loop nest locality using spec'95 and the perfect benchmarks. ACM Transactions on Computer Systems (TOCS), 17(4) 288–336, 1999.
2. V. Kathail, M. S. Schlansker, and B. R. Rau.: HPL-PD architecture specification: Version 1.1. Technical Report HPL-93-80 (R.1), Hewlett-Packard, February 2000.
3. IA-64 Application Developer's Architecture Guide, May 1999.
4. Michael E. Wolf and Monica S. Lam.: A data locality optimizing algorithm. In Proc. Of SIGPLAN PLDI '91, pages 30–44, Toronto, Ont., Jun. 1991.

5. Guang R. Gao, Vivek Sarkar, and Shaohua Han.: Locality analysis for distributed sharedmemory multiprocesors. In Proc. of the 1996 International Workshop on Languages and Compilers for Parallel Computing(LCPC), San Jose, California, Aug 1996.
6. Somnath Ghosh, Margaret Martonosi, and Sharad Malik.: Cache miss equations: An analytical representation of cache misses. In Conf. Proc., 1997 Intl. Conf. on Supercomputing,pages 317–324, Vienna, Austria, Jul. 1997.
7. The Trimaran Compiler Research Infrastructure. www.trimaran.org
8. Dinero IV Trace-Driven Uniprocessor Cache Simulator. http://www.cs.wisc.edu/~markhill/DineroIV
9. G. Rivera and C.-W. Tseng.: Eliminating conflict misses for high performance architectures. In ACM Internacional Conference on Supercomputing (ICS'98),1998
10. P.Jain, S.Devadas, D.Engels, and L.Rudolph.: Software-assisted replacement mechanisms for embedded systems. In International Conference on Computer Aided Design, pages 119-126, nov 2001.
11. Z.Wang, K.McKinley, A.Rosenberg, and C.Weems.: Using the compiler to improve cache replacement decisions.In PACT'02, September 2002.
12. Hongbo Yang, R. Govindarajan, Guang R. Gao, and Ziang Hu.: Compiler-assisted cache replacement: Problem formulation and performance evaluation. In Proceedings of the 16th International Workshop on Languages and Compilers for Parallel Compuing (LCPC’03), College Station, Texas, Oct 2003.
13. Kristof Beyls and Erik H.D’Hollander.: Compile-Time Cache Hint Generation for EPIC Architectures. In Proceedings of the 2nd International Workshop on Explicitly Parallel Instruction Computing (EPIC) Architectures and Compiler Techniques, Istanbul, Turkey, November 2002.
14. Dennis Gannon, William Jalby, and Kyle Gallivan.: Strategies for cache and local memory management by global programming transformation. Journal of Parallel and Distributed Computing, 5(5):587–616, October 1988.
15. W. Pugh.: Counting solutions to Presburger formulas: How and why. ACM SIGPLAN Notices, 29(6):121-134, jun 1994.

Modeling the Cache Behavior of Codes with Arbitrary Data-Dependent Conditional Structures*

Diego Andrade, Basilio B. Fraguela, and Ramón Doallo

Universidade da Coruña
Dept. de Electrónica e Sistemas
Facultade de Informática
Campus de Elviña, 15071 A Coruña, Spain
{dcanosa,basilio,doallo}@udc.es

Abstract. Analytical modeling is one of the most interesting approaches to evaluate the memory hierarchy behavior. Unfortunately, models have many limitations regarding the structure of the code they can be applied to, particularly when the path of execution depends on conditions calculated at run-time that depend on the input or intermediate data. In this paper we extend in this direction a modular analytical modeling technique that provides very accurate estimations of the number of misses produced by codes with regular access patterns and structures while having a low computing cost. Namely, we have extended this model in order to be able to analyze codes with data-dependent conditionals. In a previous work we studied how to analyze codes with a single and simple conditional sentence. In this work we introduce and validate a general and completely systematic strategy that enables the analysis of codes with any number of conditionals, possibly nested in any arbitrary way, while allowing the conditionals to depend on any number of items and atomic conditions.

1 Introduction

Memory hierarchies try to cushion the increasing gap between the processor and the memory speed. Fast and accurate methods to evaluate the performance of the memory hierarchies are needed in order to guide the compiler in choosing the best transformations and parameters for them when trying to make the optimal usage of this hierarchy. Trace-driven simulation [1] was the preferred approach to study the memory behavior for many years. This technique is very accurate, but its high computational cost makes it unsuitable for many applications. This way, analytical modeling, which requires much shorter computing times than previous approaches and provides more information about the reasons for the predicted

* This work has been supported in part by the Ministry of Science and Technology of Spain under contract TIC2001-3694-C02-02, and by the Xunta de Galicia under contract PGIDIT03-TIC10502PR.

P.-C. Yew and J. Xue (Eds.): ACSAC 2004, LNCS 3189, pp. 44–57, 2004.

behavior, has gained importance in recent years [2,3,4]. Still, it has important drawbacks like the lack of modularity in some models, and the limited set of codes that they can model.

In this work we present an extension to an existing analytical model that allows to analyze codes with any kind of conditional sentences. The model was already improved in [5] to enable it to analyze codes with a reference inside a simple and single conditional sentence. We now extend it to analyze codes with any kind and number of conditional sentences, even with references controlled by several nested conditionals, and nested in any arbitrary way. Like in previous works, we require the verification of the conditions in the `IF` statements to follow an uniform distribution.

This model is built around the idea of the Probabilistic Miss Equations (PMEs) [2]. These equations estimate analytically the number of misses generated by a given code in set-associative caches with LRU replacement policy. The PME model can be applied both to perfectly nested loops and imperfectly nested loops, with one loop per nesting level. It allows several references per data structure and loops controlled by other loops. Loop nests with several loops per level can also be analyzed by this model, although certain conditions need to be fulfilled in order to obtain accurate estimations. This work is part of an ongoing research line whose aim is to build a compiler framework [6,7], which extracts information from the analytical modeling, in order to optimize the execution of complete scientific codes.

The rest of the paper is organized as follows. The next section presents some important concepts to understand the PME model and our extension. Section 3 describes the process of formulation after adapting the previous existing model to the new structures it has to model. In Sect. 4 we describe the process of validation of our model, using codes with several conditional sentences. A brief review of the related work is presented in Sect. 5, followed by our conclusions and a discussion on the future work in Sect. 6.

2 Introduction to the PME Model

The Probabilistic Miss Equations (PME) model, described in [2], generates accurately and efficiently cache behavior predictions for codes with regular access patterns. The model classifies misses as either compulsory or interference misses. The former take place the first time that the lines are accessed, while the latter are associated to new accesses for which the corresponding cache line has been evicted since its previous access. The PME model builds an equation for each reference and nesting level that encloses the reference. This equation estimates the number of misses generated by the reference in that loop taking into account both kinds of misses. Its probabilistic nature comes from the fact that interference misses are estimated through the computation of a miss interference probability for every attempt of reuse of a line.

2.1 Area Vectors

The miss probability when attempting to reuse a line depends on the cache footprint of the regions accessed since the immediately preceding reference to the considered line. The PME model represents these footprints by means of what it calls *area vectors.* Given a data structure V and a k-way set-associative cache, $S_V = S_{V_0}, S_{V_1}, \ldots, S_{V_k}$ is the area vector associated with the access to V during a given period of the program execution. The i-th element, $i > 0$, of this vector represents the ratio of sets that have received $k - i$ lines from the structure; while S_{V_0} is the ratio of sets that have received k or more lines.

The PME model analyzes the access pattern of the references for each different data structure found in a program and derives the corresponding area vectors from the parameters that define those access patterns. The two most common access patterns found in the kind of codes we intend to model are the sequential access and the access to groups of elements separated by a constant stride. See [2] for more information about how the model estimates the area vectors from the access pattern parameters.

Due to the existence of references that take place with a given probability in codes with data-dependent conditionals, a new kind of access pattern arises in them that we had not previously analyzed. This pattern can ben described as an access to groups of consecutive elements separated by a constant stride, in which every access happens with a given fixed probability. The calculation of the area vector associated to this new access pattern is not included in this paper because of size limitations. This pattern will be denoted as $R_{rl}(M, N, P, p)$, which represents the access to M groups of N elements separated by a distance P where every access happens with a given probability p (see example in Sect. 4).

Very often, several data structures are referenced between two accesses to the same line of a data structure. As a result, a mechanism is needed to calculate the global area vector that represents as a whole the impact on the cache of the accesses to several structures. This is achieved by adding the area vectors associated to the different data structures. The mechanism to add two area vectors has also been described in [2], so although it is used in the following sections, we do not explain it here. The addition algorithm treats the different ratios in the area vectors as independent probabilities, thus disregarding the relative positions of the data structures in memory. This is in fact an advantage of the model, as in most situations these addresses are unknown at compile time (dynamically allocated data structures, physically indexed caches, etc.). This way, the PME model is still able to generate reasonable predictions in these cases, as opposed to most of those in the bibliography [3,4], which require the base addresses of the data structures in order to generate their estimations. Still, when such positions are known, the PME model can estimate the overlapping coefficients of the footprints associated with the accesses to each one of the structures involved, so they can be used to improve the accuracy of the addition.

```
DO I0=1, N0, L0
  DO I1=1, N1, L1
    ...
    IF cond(D(fD1(ID1), ..., fDdD(IDdD)))
      ...
      DO IZ=1, NZ, LZ
        A(fA1(IA1), ..., fAdA(IAdA))
        ...
        IF cond(B(fB1(IB1), ..., fBdB(IBdB)))
          C(fC1(IC1), ..., fCdC(ICdC))
          ...
        END IF
        ...
      END DO
      ...
    END IF
    ...
  END DO
END DO
```

Fig. 1. Nested loops with several data-dependent conditions

2.2 Scope of Application

The original PME model in [2] did not support the modeling of codes with any kind of conditionals. Figure 1 shows the kind of codes that it can analyze after applying our extension. The figure shows several nested loops that have a constant number of iterations known at compile time. Several references, which need not be in the innermost nesting level, are found in the code. Some references are affected by one or more nested conditional sentences that depend on the data arrays. All the structures are indexed using affine functions of the loop indexes $f_{A1}(I_{A1}) = \alpha_{A1} I_{A1} + \delta_{A1}$. We assume also that the verification of the conditions in the `IF` statements follows an uniform distribution, although the different conditions may hold with different probabilities. Such probabilities are inputs to our model that are obtained either by means of profiling tools, or knowledge of the behavior of the application. We assume also the conditions are independent.

As for the hardware, the PME model is oriented to set-associative caches with LRU replacement policy. In what follows, we will refer to the total size of this cache as C_s, to the line size as L_s, and k will be the degree of associativity or number of lines per set.

3 Miss Equations

The PME model estimates the number of misses generated by a code using the concept of miss equation. Given a reference, the analysis of its behavior begins in the innermost loop containing it, and proceeds outwards. In this analysis, a probabilistic miss equation is generated for each reference and in each nesting level that encloses it following a series of rules.

We will refer as $F_i(R, \text{RegInput}, \vec{p})$ to the miss equation for reference R in nesting level i. Its expression depends on RegInput, the region accessed since the last access to a given line of the data structure. Since we now consider the existence of conditional sentences, the original PME parameters have been extended with a new one, $\vec{p}$. This vector contains in position j the probability p_j that the (possible) conditionals that guard the execution of the reference R in nesting level j are verified. If no conditionals are found in level j, then $p_j = 1$. When there are several nested `IF` statements in the same nesting level, p_j corresponds to the product of their respective probabilities of holding their respective conditions. This is a first improvement with respect to our previous approach [5], which used a scalar because only a single conditional was considered.

Depending on the situation, two different kinds of formulas can be applied:

- If the variable associated to the current loop i does not index any of the references found in the condition(s) of the conditional(s) sentence(s), then we apply a formula from the group of formulas called *Condition Independent Reference Formulas* (CIRF). This is the kind of PME described in [2].
- If the loop variable indexes any of such references, then we apply a formula from the group called *Condition Dependent Reference Formulas* (CDRF).

Another factor influencing the construction of a PME is the existence of other references to same data structure, as they may carry some kind of group reuse. For simplicity, in what follows we will restrict our explanation to references that carry no reuse with other references.

3.1 Condition Independent Reference Formulas

When the index variable for the current loop i is not among those used in the indexing of the variables referenced in the conditional statements that enclose the reference R, the PME for this reference and nesting level is given by

$$\begin{aligned} F_i(R, \text{RegInput}, \vec{p}) =& L_{Ri} F_{i+1}(R, \text{RegInput}, \vec{p}) \\ &+ (N_i - L_{Ri}) F_{i+1}(R, \text{Reg}(A, i, 1), \vec{p}) \ , \end{aligned} \tag{1}$$

being N_i the number of iterations in the loop of the nesting level i, and L_{Ri} the number of iterations in which there is no possible reuse for the lines referenced by R. $\text{Reg}(A, i, j)$ stands for the memory region accessed during j iterations of the loop in the nesting level i that can interfere with data structure A.

The formula calculates the number of misses for a given reference R in nesting level i, as the sum of two values. The first one is the number of misses produced by the L_{Ri} iterations in which there can be no reuse in this loop. The miss probability for these iterations depends on the accesses and reference pattern in the outer loops. The second value corresponds to the iterations in which there can be reuse of cache lines accessed in the previous iteration, and so it depends on the memory regions accesses during one iteration of the loop.

The indexes of the reference R are affine functions of the variables of the loops that enclose it. As a result, R has a constant stride S_{Ri} along the iterations of loop i. This value is calculated as $S_{Ri} = \alpha_{A_j} d_{A_j}$, where j is the dimension whose index depends on $\mathtt{I}_i$, the variable of the loop; α_{A_j} is the scalar that multiplies the loop variable in the affine function, and d_{A_j} is the size of the j-th dimension. If I_i does not index reference R, then $S_{Ri} = 0$. This way, L_{Ri} can be calculated as,

$$L_{Ri} = 1 + \left\lfloor \frac{N_i - 1}{max\{L_s/S_{Ri}, 1\}} \right\rfloor \ , \tag{2}$$

The formula calculates the number of accesses of R that cannot exploit either spatial or temporal locality, which is equivalent to estimating the number of different lines that are accessed during N_i iterations with stride S_{Ri}.

3.2 Condition Dependent Reference Formulas

If the index variable for the curret loop i is used in the indexes of the arrays used in the conditions that control the reference R, the behavior of R with respect to this loop is irregular. The reason is that different values of the index access different pieces of data to test in the conditions. This way, in some iterations the conditions hold and R is executed, thus affecting the cache, while in other iterations the associated conditions do not hold and no access of R takes place. As a result, the reuse distance for the accesses of R is no longer fixed: it depends on the probabilities $\vec{p}$ that the conditions that control the execution of R are verified. If the probabilities the different conditions hold are known, the number of misses associated to the different reuse distances can be weighted using the probability each reuse distante takes place.

As we have just seen, eq. (2) estimates the number L_{Ri} of iterations of the loop in level i in which reference R cannot explote reuse. Since the loop has N_i iterations, this means on average each different line can be reused in up to $G_{Ri} = N_i/L_{Ri}$ consecutive iterations. Besides, either directly reference R or the loop in level $i+1$ that contains it can be inside a conditional in level i that holds with probability p_i. Thus, $p_i L_{Ri}$ different groups of lines will be accessed on average, and each one of them can be reused up to G_{Ri} times. Taking this into account, the general form of a condition-dependent PME is

$$F_i(R, \text{RegInput}, \vec{p}) = p_i L_{Ri} \sum_{j=1}^{G_{Ri}} \text{WMR}_i(R, \text{RegInput}, j, \vec{p}) \ . \tag{3}$$

where $\text{WMR}_i(\text{RegInput}, j, \vec{p})$ is the weighted number of misses generated by reference R in level i considering the j-th attempt of reuse of the G_{Ri} ones potentially possible. As in Sect. 3.1, RegInput is the region accessed since the last access to a given line of the considered data structure when the execution of the loop begins. Notice that if no condition encloses R or the loop around it in this level, simply $p_i = 1$.

The number of misses associated to reuse distance j weigthed with the probability an access with such reuse distante does take place, is calculated as

$$\begin{aligned}\mathrm{WMR}_i(\mathrm{RegInput}, j, \vec{p}) =& (1 - P_i(R, \vec{p}))^{j-1} F_{i+1}(R, \mathrm{RegInput} \cup \mathrm{Reg}(A, i, j-1), \vec{p}) + \\ & \sum_{k=1}^{j-1} P_i(R, \vec{p})(1 - P_i(R, \vec{p}))^{k-1} F_{i+1}(R, \mathrm{Reg}(A, i, k-1), \vec{p}) \ ,\end{aligned} \tag{4}$$

where $P_i(R, \vec{p})$ yields the probability that R accesses each of the lines it can potentially reference during one iteration of the loop in nesting level i. This probability is a function of those conditionals in $\vec{p}$ in or below the nesting level analyzed. The first term in (4) considers the case that the line has not been accessed during any of the previous $j-1$ iterations. In this case, the RegInput region that could generate interference with the new access to the line when the execution of the loop begins must be added to the regions accessed during these $j-1$ previous iterations of the loop in order to estimate the complete interference region. The second term weights the probability that the last access took place in each of the $k = 1, \ldots, j-1$ previous iterations of the considered loop.

Line Access Probability. The probability $P_i(R, \vec{p})$ that the reference R whose behavior is being analyzed does access one of the lines that belong to the region that it can potentially access during one iteration of loop i is a basic parameter to derive $\mathrm{WMR_i}(\mathrm{RegInput}, j, \vec{p})$, as we have just seen. This probability depends not only on the access pattern of the reference in this nesting level, but also in the inner ones, so its calculation takes into account all the loops from the i-th down to the one containing the reference. If fact, this probability is calculated recursively in the following way:

$$P_i(R, \vec{p}) = \begin{cases} p_i & \text{if } i \text{ is the innermost loop} \\ & \text{that contains } R \\ p_i P_{i+1}(R, \vec{p}) & \text{if the index of loop } i+1 \text{ is} \\ & \text{not used in the references} \\ & \text{in conditions that control } R \\ p_i(1 - (1 - P_{i+1}(R, \vec{p}))^{G_{R_{i+1}}}) & \text{otherwise} \end{cases} \tag{5}$$

where we must remember that p_i is the product of all the probabilities associated to the conditional sentences affecting R that are located in nesting level i.

This algorithm to estimate the probability of access per line at level i has been improved with respect to our previous work [5], as it is now able to integrate different conditions found in different nesting levels, while the previous one only considered a single condition.

```
posB=1
DO I=1,N
   offB(I)=posB
   DO J=1,M
      IF A(I,J).NEQ.0
         B(posB)=A(I,J)
         jB(posB)=J
         posB=posB+1
      ENDIF
   ENDDO
ENDDO
```

Fig. 2. CRS Storage Algorithm

```
DO I=1,M
  DO K=1,N
    IF A(I,K).NEQ.0
      DO J=1,P
        IF B(K,J).NEQ.0
          C(I,J)=C(I,J)+A(I,K)*B(K,J)
        ENDIF
      ENDDO
    ENDIF
  ENDDO
ENDDO
```

Fig. 3. Optimized product of matrices

3.3 Calculation of the Number of Misses

In the innermost level that contains the reference R, both in CIRFs and CDRFs, $F_{i+1}(R, \mathrm{RegInput}, \vec{p})$, the number of misses caused by the reference in the immediately inner level is $AV_0(\mathrm{RegInput})$, this is, the first element in the area vector associated to the region RegInput.

The number of misses generated by reference R in the analyzed nest is finally estimated as $F_0(R, \mathrm{RegInput}_{\mathrm{total}}, \vec{p})$ once the PME for the outermost loop is generated. In this expression, $\mathrm{RegInput}_{\mathrm{total}}$ is the total region, this is, the region that covers the whole cache. The miss probability associated with this region is one.

4 Model Validation

We have validated our model by applying it manually to the two quite simple but representative codes shown in Fig. 2 and Fig. 3. The first code implements the storage of a matrix in CRS format (Compressed Row Storage), which is widely used in the storage of sparse matrices. It has two nested loops and a conditional sentence that affects three of the references. The second code is an optimized product of matrices; that consists of a nest of loops that contain references inside several nested conditional sentences.

Results for both codes will be shown in Sect. 4.2, but we will first focus on the second code in order to provide a detailed idea about the modeling procedure.

4.1 Optimized Product Modeling

This code is shown in Fig. 3. It implements the product between two matrix, A and B, with a uniform distribution of nonzero entries. As a first optimization, when the element of `A` to be used in the current product is 0, then all its products with the corresponding elements of `B` are not performed. As a final optimization, if the element of `B` to be used in the current product is 0 then that operation is not performed. This avoids two floating point operations and the load and storage of C(I,J).

Without loss of generality, we assume a compiler that maps scalar variables to registers and which tries to reuse the memory values recently read in processor registers. Under these conditions, the code in Fig. 3 contains three references to memory. The model in [2] can estimate the behavior of the references `A(I,K)`, which take place in every iteration of their enclosing loops.

Thus, we will focus our explanation on the modeling of the behavior of the references `C(I,J)` and `B(K,J)` which are not covered in previous publications.

`C(I,J)` Modeling. The analysis begins in the innermost loop, in level 2. In this level the loop variable indexes one of the reference of one of the conditions, so the CDRF formula must be applied.

As $S_{R2} = P$, $L_{R2} = 1 + N$, $G_{R2} \simeq 1$ and p_2 is the component in vector $\vec{p}$ associated to the probability that the condition inside the loop in nesting level 2 holds. This loop is in the innermost level. Thus, $F_3(R, \mathrm{RegInput}, \vec{p}) = AV_0(\mathrm{RegInput})$, then after the simplification the formulation is,

$$F_2(R, \mathrm{RegInput}, \vec{p}) = p_2 P AV_0(\mathrm{RegInput}) \ . \tag{6}$$

In the next upper level, level 1, the loop variable indexes one reference of one of the conditions, so the CDRF formula has to be applied. Let $S_{R1} = 0$, $L_{R1} = 1$ and $G_{R1} \simeq N$, then

$$F_1(R, \mathrm{RegInput}, \vec{p}) = p_1 \sum_{j=1}^{N} \mathrm{WMR}_1(R, \mathrm{RegInput}, j, \vec{p}) \ . \tag{7}$$

In order to compute WMR_1 we need to calculate the value for two functions. One is $P_1(R, \vec{p})$, which for our reference takes the value $p_1 p_2$, where p_i is the i-th element in vector $\vec{p}$. The other one is $\mathrm{Reg}(C, 1, i)$, the region accessed during i iterations of the loop 1 that can interfere with the accesses to `C`.

$$\begin{aligned}\mathrm{Reg}(C,1,i) =& R_{rl_{\mathrm{auto}}}(P,1,M,1-(1-p_1p_2)^i) \\ & \cup R_r(i,1,M) \cup R_{rl}(P,i,N,p_1)\ .\end{aligned} \tag{8}$$

The first term is associated to the autointerference of C, which is the access to P groups of one element separated by a difference M and every access takes places with a given probability. The second term represents the access to i groups of 1 element separated by a distance M. The last element represents the access to P groups of i elements separated by a distance N. Every access is going to happen with a given probability p_1.

In the outermost level, the loop variable indexes the reference of the condition. As a result, the CDRF formula is to be applied again. Being $S_{R0} = 1$, $L_{R0} = 1 + \lfloor (N-1)/L_s \rfloor$ and $G_{R0} \simeq L_s$, so the formulation is

$$F_0(R, \mathrm{RegInput}, \vec{p}) = (1 + \lfloor (N-1)/L_s \rfloor) \sum_{j=1}^{L_s} \mathrm{WMR}_0(R, \mathrm{RegInput}, j, \vec{p})\ . \tag{9}$$

As before, two functions must be evaluated to compute WMR_0. They are $P_0(R,\vec{p}) = 1-(1-p_1p_2)^N$ and $\mathrm{Reg}(C,0,i)$, given by

$$\begin{aligned}\mathrm{Reg}(C,0,i) =& R_{rl_{\mathrm{auto}}}(P,1,M,1-(1-p_1p_2)^N) \\ & \cup R_r(N,i,M) \cup R_l(PN, 1-(1-p_1)^{L_s})\ .\end{aligned} \tag{10}$$

The first term is associated to the autointerference of C, which is the access to P groups of one element separated by a difference M and every access takes places with a given probability. The second term represents the access to N groups of i elements separated by a distance M. The last element represents the access to PN consecutive elements with a given probability.

B(K,J) Modeling. The innermost loop for this reference is also the one in level 2. The variable that controls this loop, J, is found in the indexes of a reference found in the condition of an IF statements (in this case, the innermost one), one conditional, so a CDRF is to be built. As this is the innermost loop, we get $F_3(R, \mathrm{RegInput}, \vec{p}) = AV_0 RegInput$. Since $S_{R_i} = N$, $L_{R_i} = P$ and $G_{R_i} = 1$ the formulation for this nesting level is

$$F_2(R, S(\mathrm{RegInput}), \vec{p}) = PAV_0(\mathrm{RegInput})\ . \tag{11}$$

The next level is level 1. In this level the variable of the loops indexes any of the reference of any of the conditional, so we have to use the CDRF formula. Being $S_{R1} = 1$, $L_{R1} = 1 + \lfloor (N-1)/L_s \rfloor$ and $G_{R1} \simeq L_s$ the formulation is

$$F_1(R, \mathrm{RegInput}, \vec{p}) = p_1 \left(1 + \left\lfloor \frac{N-1}{L_s} \right\rfloor\right) \sum_{j=1}^{L_s} \mathrm{WMR}_1(R, \mathrm{RegInput}, j, \vec{p})\ . \tag{12}$$

We need to know $P_1(R,\vec{p}) = p_1$ and the value of the accessed regions $\mathrm{Reg}(B,1,i)$ in order to compute WMR_1:

$$\mathrm{Reg}(B,1,i) = Rrl_{\mathrm{auto}}(P,i,N,p_1]) \cup R_r(i,1,M) \cup R_{rl}(P,1,M,p_1p_2)\ . \tag{13}$$

Table 1. Validation data for the code in Fig. 2 for several cache configurations and different problem sizes and condition probabilities

M	N	p	C_s	L_s	K	Δ_{MR}	T_{sim}	T_{exe}	T_{mod}
6200	10150	0.4	32K	8	4	0.001	82	19	0.001
4200	17150	0.1	4K	4	2	0.401	107	18	0.001
16220	7200	0.2	16K	4	2	2.635	152	24	0.044
6200	14250	0.3	32K	8	4	0.005	146	22	0.001
9200	14250	0.1	4K	4	8	2.374	582	50	0.001
1100	15550	0.5	4K	4	8	0.027	2	1	0.001
2900	17250	0.3	32K	16	4	1.847	65	32	0.001
8900	9250	0.1	64K	8	4	3.055	118	46	0.010
4200	12150	0.1	4K	4	2	0.571	64	33	0.001
5000	15000	0.3	32K	8	4	0.183	125	54	0.001
7200	12250	0.1	4K	4	8	0.044	139	64	0.010

The first term is associated to the autointerference of B, which is the access to P groups of i elements separated by a difference N and every access takes places with a given probability. The second term represents the access to i groups of one element separated by a distance M. The last element represents the access to P groups of one element separated by a distance M, every access takes places with a given probability $p_1 p_2$.

In the outermost level, the level 0, the variable of the loop indexes a reference in one of the conditions, so we have to apply again the CDRF formula. Being $S_{R0} = 0$, $L_{R0} = 1$, $G_{R0} \simeq M$, so the formulation is

$$F_0(R, \text{RegInput}, \vec{p}) = \sum_{j=1}^{M} \text{WMR}_0(R, \text{RegInput}, j, \vec{p}) \ . \tag{14}$$

In this loop, WMR_0 is a function of $P_0(R, \vec{p} = 1 - (1 - p_1)^{L_s}$ and the value of the accessed regions $\text{Reg}(B, 0, i)$:

$$\begin{aligned} \text{Reg}(B, 0, i) = & R_{l_{\text{auto}}}(PN, 1 - (1 - p_1)^{L_s}) \cup R_r(N, i, M) \\ & \cup R_{rl}(P, i, M, 1 - (1 - p_1 p_2)^N) \ . \end{aligned} \tag{15}$$

The first term is associated to the autointerference of B, which is the access to PN elements with a given probability. The second term represents the access to N groups of i elements separated by a distance M. The last element represents the access to P groups of i elements separated by a distance M, every access takes places with a given probability.

4.2 Validation Results

We have done the validation by comparing the results of the predictions given by the model with the results of a trace-driven simulation. We have tried several cache configurations, problem sizes and probabilities for the conditional sentences.

Table 2. Validation data for the code in Fig. 3 for several cache configurations and different problem sizes and condition probabilities

M	N	P	p_1	p_2	C_s	L_s	K	Δ_{MR}	T_{sim}	T_{exe}	T_{mod}
750	750	1000	0.2	0.1	16K	8	8	0.808	35	17	0.075
750	750	1000	0.8	0.3	8K	16	16	5.081	164	62	0.053
900	850	900	0.8	0.1	16K	32	2	0.224	78	54	0.795
900	850	900	0.9	0.1	64K	8	8	0.589	136	66	0.523
900	950	1500	0.8	0.3	16K	4	2	2.411	236	159	0.110
900	950	1500	0.1	0.4	32K	8	4	5.408	51	38	0.357
1000	850	900	0.7	0.5	4K	8	2	4.394	98	97	0.054
200	250	150	0.8	0.2	16K	4	2	0.604	1	0	0.690
200	250	150	0.1	0.3	32K	8	4	2.161	0	0	0.145
200	250	150	0.3	0.1	4K	4	8	1.208	0	0	0.008
100	350	90	0.8	0.5	4K	4	8	0.070	0	1	0.042
100	350	90	0.4	0.4	8K	8	4	0.417	0	0	0.324
100	350	90	0.2	0.3	4K	8	2	0.744	0	0	0.218

Tables 1 and 2 display the validation results for the codes in Fig. 2 and 3, respectively. In Table 1 the two first columns contain the problem size and the third column stands for the probability p that the condition in the code is fulfilled. In Table 2 the first three columns contain the problem size, while the next two columns contain the probabilities p_1 and p_2 that each of of the two conditions in Fig. 3 is fulfilled. Then the cache configuration is given in both tables by C_s, the cache size, L_s, the line size, and the degree of associativity of the cache, K. The sizes are measured in the number of elements of the arrays used in the codes. The accuracy of the model is used by the metric Δ_{MR}, which is based on the miss rate (MR); it stands for the absolute value of the difference between the predicted and the measured miss rate.

For every combination of cache configuration, problem size and probabilities of the conditions, 25 different simulations have been made using different base addresses for the data structures.

The results show that the model provides a good estimation of the cache behavior in the two example codes. The last three columns in both tables reflect the corresponding simulation times, source code execution time and modeling times expressed in seconds and measured in a 2,08 Ghz AMD K7 processor-based system. We can see that the modeling times are much smaller than the trace-driven simulation and even execution times. Furthermore, modeling times are several orders of magnitude shorter than trace-driven simulation and even execution times. The modeling time does not include the time required to build the formulas for the example codes. This will be made automatically by the tool we are currently developing. According to our experience in [2], the overhead of such tool is negligible.

5 Related Work

Over the years, several analytical models have been proposed to study the behavior of caches. Probably the most well-known model of this kind is [8], based on the Cache Miss Equations (CMEs), which are lineal systems of Diophantine equations. Its main drawbacks are its high computational cost and that it is restricted to analyzing regular access patterns that take place in isolated perfectly nested loops. In the past few years, some models that can overcome some of these limitations have arisen. This is the case of the accurate model based on Presburger formulas introduced in [3], which can analyze codes with non-perfectly nested loops and consider reuses between loops in different nesting levels. Still, it can only model small levels of associativity and it has a extremely high computational cost. More recently [4], which is based on [8], can also analyze these kinds of codes in competitive times thanks to the statistical techniques it applies in the resolution of the CMEs.

A more recent work [9], can model codes with conditional statements. Still, it does not consider conditions on the input or intermediate data computed by the programs. It is restricted to conditional sentences whose conditions refer to the variables that index the loops.

All these models and others in the bibliography have fundamental differences with ours. One of the most important ones is that all of them require a knowledge about the base address of the data structures. In practice this is not possible or useful in many situations because of a wide variety of reasons: data structures allocated at run-time, physically-indexed caches, etc. Also, thanks to the general strategy described in this paper, the PME model becomes the first one to be able to model codes with data-dependent conditionals.

6 Conclusions and Future Work

In this work we have presented an extension to the PME model described in [2]. The extension allows this model to be the first one that can analyze codes with data-dependent conditionals and considering, not only simple conditional sentences but also nested conditionals affecting a given reference. We are currently limited by the fact that the conditions must follow an uniform distribution, but we think our research is an important step in the direction of broadening the scope of applicability of analytical models. Our validation shows that this model provides accurate estimations of the number of misses generated by a given code while requiring quite short computing times. In fact the model is typically two orders of maginute faster than the native execution of the code.

The properties of this model turn it into an ideal tool to guide the optimization process in a production compiler. In fact, the original PME model has been used to guide the optimization process in a compiler framework [7]. We are now working in an automatic implementation of the extension of the model described in this paper in order to integrate it in that framework. As for the scope of the program structures that we wish to be amenable to analysis using the PME model, our next step will be to consider codes with irregular accesses due to the use of indirections or pointers.

References

1. Uhlig, R., Mudge, T.: Trace-Driven Memory Simulation: A Survey. ACM Computing Surveys **29** (1997) 128–170
2. Fraguela, B.B., Doallo, R., Zapata, E.L.: Probabilistic Miss Equations: Evaluating Memory Hierarchy Performance. IEEE Transactions on Computers **52** (2003) 321–336
3. Chatterjee, S., Parker, E., Hanlon, P., Lebeck, A.: Exact Analysis of the Cache Behavior of Nested Loops. In: Proc. of the ACM SIGPLAN'01 Conference on Programming Language Design and Implementation (PLDI'01). (2001) 286–297
4. Vera, X., Xue, J.: Let's Study Whole-Program Behaviour Analytically. In: Proc. of the 8th Int'l Symposium on High-Performance Computer Architecture (HPCA8). (2002) 175–186
5. Andrade, D., Fraguela, B., Doallo, R.: Cache behavior modeling of codes with data-dependent conditionals. In Springer-Verlag, ed.: 7th Intl. Workshop on Software and Compilers for Embedded Systems, SCOPES 2003. Volume 2826 of Lecture Note in Computer Science. (2003) 373–387
6. Blume, W., Doallo, R., Eigenmann, R., Grout, J., Hoeflinger, J., Lawrence, T., Lee, J., Padua, D., Paek, Y., Pottenger, B., Rauchwerger, L., Tu, P.: Parallel Programming with Polaris. IEEE Computer **29** (1996) 78–82
7. Fraguela, B.B., Touri o, J., Doallo, R., Zapata, E.L.: A compiler tool to predict memory hierarchy performance of scientific codes. Parallel Computing **30** (2004) 225–248
8. Ghosh, S., Martonosi, M., Malik, S.: Cache Miss Equations: A Compiler Framework for Analyzing and Tuning Memory Behavior. ACM Transactions on Programming Languages and Systems **21** (1999) 702–745
9. Vera, X., Xue, J.: Efficient Compile-Time Analysis of Cache Behaviour for Programs with IF Statements. In: 5th International Conference on Algorthms and Archiectures for Parallel Processing. (2002) 396–407

A Configurable System-on-Chip Architecture for Embedded Devices

Sebastian Wallner

Department of Distributed Systems, Technical University Hamburg-Harburg,
Schwarzenbergstrasse 95, D-21073 Hamburg, Germany
{wallner@tu-harburg.de}

Abstract. This paper describes a novel Configurable System-on-Chip (CSoC) architecture for stream-based computations and real-time signal processing. It offers high computational performance and a high degree of flexibility and adaptability by employing a micro Task Controller (mTC) unit in conjunction with programmable and configurable hardware. The hierarchically organized architecture provides a programming model, allows an efficient mapping of applications and is shown to be easy scalable to future VLSI technologies where over a hundred processing cells on a single chip will be feasible to deal with the inherent dynamics of future application domains and system requirements. Several mappings of commonly used digital signal processing algorithms and implementation results are given for a standard-cell ASIC design realization in 0.18 micron 6-layer UMC CMOS technology.

1 Introduction

We are currently experiencing an explosive growth in development and deployment of embedded devices such as multimedia set-top-boxes and personal mobile computing systems which demands an increasing support of multiple standards [1,2]. This flexibility requirement points to the need of various communication, audio and video algorithms which differ in complexity. They have mostly a heterogenous nature and comprise several sub-tasks with real-time performance requirements for data-parallel tasks [3]. 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 needs different levels of control over the functional units and different memory access. For instance, multimedia applications (like different video decompression schemes) may include a data-parallel task, a bit-level task, irregular computations, high-precision word operations and a real-time component [4]. The addressed requirements becomes even more relevant when Quality-of-Service (QoS) requirements e.g. varying the communication bandwidth in wireless terminals, variable audio quality or a change from full color to black/white picture quality becomes a more important factor. A way to solve the flexibility and adaptability demands has been to use General Purpose Processors (GPP) or Digital Signal Processors (DSP), i.e. trying to solve all kinds of applications

P.-C. Yew and J. Xue (Eds.): ACSAC 2004, LNCS 3189, pp. 58–71, 2004.

running on a very high speed processor. A major drawback of using these general-purpose devices is that they are extremely inefficient in terms of utilizing their resources to best take advantage of data-level parallelism in the algorithms. Todays demands motivate the use of hybrid architectures which integrate programmable logic together with different embedded resources and Configurable Systems-on-Chip which can be realized by integrating reconfigurable (re-usable) and programmable hardware components. In contrast to processors, they totally lack programming models that would allow for device independent compilation and forward compatibility to other architecture families.

The approach in this paper describes a Configurable System-on-Chip approach with configurable and programmable properties. The architecture combines a wide variety of macro-module resources including a MIPS-like scalar processor core, coarse-grained reconfigurable processing arrays, embedded memories and custom modules supervised by a micro Task Controller. In the architecture, functions can be dynamically assigned to physical hardware resources such that the most efficient computation can be obtained. It can be forward compatible to other CSoC families with variable numbers of reconfigurable processing cells for different performance features. A major key issue for the CSoC system integration includes the coupling of the macro-module resources for efficient mapping and transfer of data. The programming aspect is another important aspect in this paper.

This work is organized as follows. Section 2 presents related work. Section 3 the reconfigurable processing array which based on previous research activities. Section 4 introduces the CSoC architecture composition and the system control mechanism in detail. The next section (section 5) presents the programming paradigm. Algorithms mapping and performance analysis are present in Section 6. Finally, section 7 discusses the design- and physical implementation while conclusions and future work are drawn in Section 8.

2 Related Work

There have been several research efforts as well as commercial products that have tried to explore the use of reconfigurable- and System-on-Chip architectures. They integrate existing components (IP-cores) into a single chip or explore complete new architectures. In the Pleiades project at UC Berkeley [5], the goal is to create a low-power high-performance DSP system. Yet, the Pleiades architecture template differs from the proposed CSOC architecture. In the Pleiades architecture a general purpose microprocessor is surrounded by a heterogeneous array of autonomous special-purpose satellite processors communicated over a reconfigurable communication network. In contrast to the Pleiades system the proposed architecture offers reconfigurable hardware and data-paths supervised by a flexible controller unit with a simple instruction set which allows conditional reconfiguration and efficient hardware virtualization. The reconfigurable architecture CS2112 Reconfigurable Communication Processor [6] from Chameleon Systems couples a processor with a reconfigurable fabric composed of 32-bit processor tiles. The fabric holds a background plane with configuration data which can be loaded while the active plane is in use. A small state-machine controls every tile. The embedded processor manages the reconfiguration and streaming data. The chameleon chip has a fixed architecture targeting communication applications. The Configurable SoC architecture offers a micro Task Controller which allows variable handling of different processing resources and provides forward compatability with other CSoC architecture families for different performance demands. The scalar processor core is not involved in the configuration process. Furthermore, the configuration mechanism differs completely from the configuration approach used by Chameleon Systems. MorphoSys from the University of California Irvine [7] has a MIPS-like ”TinyRISC” processor with extended instruction set, a mesh-connected 8 by 8 reconfigurable array of 28 bit ALUs. The ”TiniRISC” controls system execution by initiating context memory and frame buffer loads using extra instructions via a DMA controller. MorphoSys offers dynamic reconfiguration with several local configuration memories. The suggested architecture model includes a micro Task Controller with a simple instruction set and a single local configuration memory in each cluster. It uses a pipelined configuration concept to configure multiple reconfigurable processing cells. The micro task program and the descriptor set can be reused in other CSoC families without a recompilation task.

3 Background

The Configurable System-on-Chip architecture approach build on previous research activities in identifying reconfigurable hardware structures and providing a new hardware virtualization concept for coarse-grained reconfigurable architectures. A reconfigurable processing cell array (RPCA) has been designed which targets applications with inherent data-parallelism, high regularity and high throughput requirements [8]. The architecture is based on a synchronous multifunctional pipeline flow model using reconfigurable processing cells and configurable data-paths. A configuration manager allows run-time- and partial reconfiguration. The RPCA consists of an array of configurable coarse-grained processing cells linked to each other via broadcast- and pipelined data buses. It is fragmented into four parallel stripes which can be configured in parallel. The configuration technique based of an pipelined configuration process via descriptors. Descriptors represent configuration templates abutted to instruction operation-codes in conventional Instruction Set Architectures (ISA). They can be sliced into fixed-size computation threads that, in analogy to virtual memory pages, are swapped onto available physical hardware within a few clock cycles. The architecture approach results in a flexible reconfigurable hardware component with performance and function flexibility. Figure 1 shows a cluster with overall 16 processing cells.

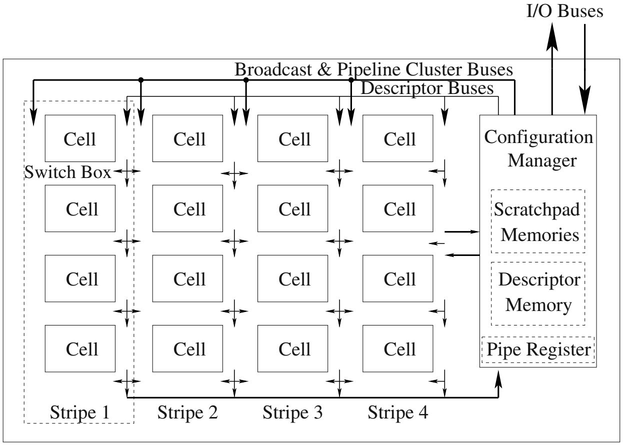


Fig. 1. The structure of a cluster with the configuration manager, the processing cells with the switch boxes for routing, the dual-port scratch pad memories and the descriptor memory. In order to adjust configuration cycles, three pipeline registers for every stripe are implemented.

Some important characteristics of the reconfigurable architecture are:

- Coarse-grained reconfigurable processing model: Each reconfigurable processing cell contains a multifunctional ALU that operates on 48-bit or $2*24$-bit data words (split ALU mode) on signed integer and signed fixed-point arithmetic.
- Computation concept via compute threads: An application is divided in several computation threads which are mapped and executed on the processing array.
- Hardware virtualization: Hardware on demand with a single context memory and unbounded configuration contexts (descriptors) with low overhead context switching for high computational density.
- Configuration data minimization: Reuse of a descriptor, to configure several processing cells, minimizes the configuration memory.
- Library-based design approach: Library contains application-specific kernel modules composed of several descriptors to reduce the developing-time and cost.
- Scalable architecture for future VLSI technologies: Configuration concept easy expandable to larger RPCA.

The RPCA is proposed to be the reconfigurable module of the Configurable System-on-Chip design. It executes the signal processing algorithms in the application domain to enhance the performance of critical loops and computation intensive functions.

4 Architecture Composition

The CSoC architecture model is hierarchically organized. It is composed of a micro Task Controller (mTC) unit which includes a set of heterogeneous processing resources and the reconfigurable processing cell array. Figure 2 gives a structure overview. To avoid that a global bus is becoming a bottleneck, a high-speed crossbar switch connects the heterogeneous processing resources and the RPCA. It allows multiple data transfers in parallel. Many algorithms in the applications domain contains abundant parallelism and are compute intensive. They are preferable spatially mapped onto the RPCA via a set of descriptors while the other parts can be computed with the heterogeneous processing resources [8]. An advantage of such a architecture partitioning is the more efficient mapping of algorithms as the Fast Fourier Transformation example in section 6 illustrates.

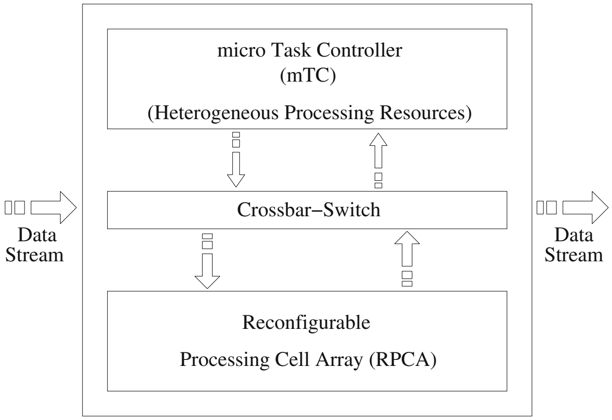


Fig. 2. The hierarchically organized CSoC structure with the heterogeneous processing resources and the reconfigurable processing cell arrays are connected via a crossbar switch. The RCPA is partitioned in several clusters.

4.1 Configurable System-on-Chip Overview

The CSoC architecture consists of the mTC and two clusters of reconfigurable processing cells. Every cluster comprises sixteen 48-bit coarse-grained reconfigurable processing cells. It offers variable parallelism and pipelining for different performance requirements. A two channel programmable DMA controller handles the data transfers between the external main memory banks and the processing cell array. Figure 3 outlines the architecture overview.

In high performance systems high memory bandwidth is mandatory. Thus the architecture uses a bandwidth hierarchy to bridge the gap between deliverable

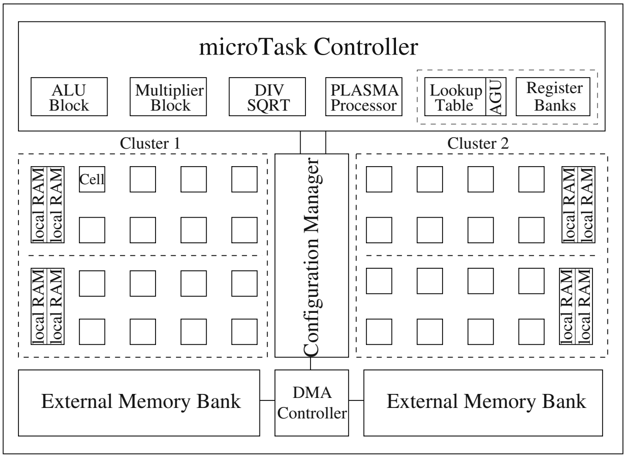


Fig. 3. The overall CSoC architecture with the mTC unit which includes the heterogeneous processing resources, two clusters of reconfigurable processing cells with the scratch-pad memories (local RAM) and the Plasma scalar processor.

off-chip memory bandwidth and the bandwidth necessary for the computation required by the application. The hierarchy has two levels: a main memory level (External Memory Bank) for large, infrequently accessed data realized as external memory block and a local level for temporary use during calculation realized as high-speed dual ported scratch-pad memories (local RAM).

4.2 Processing Resource Implementation

Due to the application field in digital signal, image and video processing applications the heterogeneous processing resources comprises

- two ALU (Arithmetic Logic Unit) blocks with four independent 48- or eight 24-bit adder units, saturation logic and boolean function,
- a multiplier block with two 24-bit signed/unsigned multipliers,
- a 24-bit division and square root unit,
- a MIPS-I compatible 32-bit Plasma scalar processor core,
- a $256 * 24$-bit lookup table with two independent Address Generation Units (AGU),
- two register banks of $4 * 48$-bit universal registers ($R0 - R3\,;\ R0' - R3'$).

The 32-bit Plasma scalar processor core can be used for general-purpose processing or for tasks which can not be very well accomplished on the other processing resources. It can directly address the local RAMs of the reconfigurable processing cells. Bit computation is achievable with the ALU block. The lookup-table and the address generation unit can be used to construct e.g. a Direct Digital Frequency Synthesis (DDFS) generator which is frequently used in telecommunication applications.

4.3 System Control Mechanism and Instruction Set

The micro Task Controller is a global micro-programmed control machine which offers a versatile degree of sharing the architecture resources. It manages the heterogeneous processing resources and controls the configuration manager without retarding the whole system. The mTC executes a sequence of micro-instructions which are located in a micro-program code memory. It has a micro-program address counter which can be simply increment by one on each clock cycle to select the address of the next micro-instruction. Additionally, the micro Task Controller provides an instruction to initiate the configuration and control-flow instructions for branching. The following four instructions are implemented:

- GOTO [Address]: Unconditional jump to a dedicated position in the micro-program code.
- FTEST [Flag, Address]: Test a flag and branch if equal; If the flag is set (e.g. saturation flag), jump to a dedicated position in the micro-program code otherwise continue.
- TEST [Value, Register, Address]: Test a 48/24-bit value and branch if equal; The instruction is equivalent to the FTEST instruction. It tests the content of an universal register.
- CONFIG [Function Address]: Start address of the configuration context in the descriptor memory; The configuration manager fetches the descriptors in addiction to the descriptor type. The configuration process must only be triggered. The configuration manager notifies the termination of the configuration process by a finish flag.

The instruction set allows to control the implemented heterogeneous processing resources, the configuration manager and the data-streams. It allows conditional reconfiguration and data dependent branching. $If-then-else$, *while* and *select* assignations can be easily modeled.

5 Programming Paradigm

An application is represented as a Control Data-Flow Graph (CDFG). In this graph, control (order of execution) and data are modeled in the same way. It conveys the potential concurrencies within an algorithm and facilitates the parallelization and mapping to arbitrary architectures. The graph nodes usually correspond to primitives, such as FFT, FIR-filter or complex multiplication (CMUL). For a given network architecture, the programmer starts with partitioning and mapping of the graph in micro-tasks (subtasks), by allocate the flow graph nodes to the reconfigurable and heterogeneous processing resources in the system. It specifies the program structure: the sequence of micro-tasks that comprises the application, the connection between them and the input and output ports for the data streams. A micro-task processes input data and produces output data. After a task finishes, its output is typically input to the next micro-task. Figure 4 shows an example composed of several micro-tasks with data feed-back.

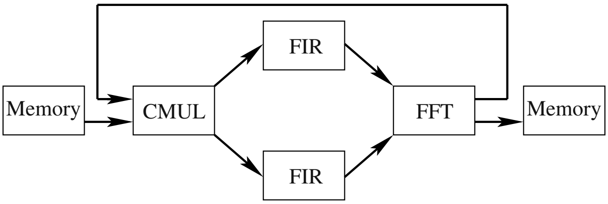


Fig. 4. An example of a cascade of computation tasks with data feedback. A task may consists of a configuration and a processing part.

After the partitioning and mapping, a scheduling process produces the schedule for the application which determines the order of the micro-task execution. The schedule for a computation flow is stored in a microprogram as a sequence of steps that directs the processing resources. It is represented as a mTC reservation table with a number of usable reservation time slots. The allocation of these time slots can be predefined by the scheduler process via a determined programming sequence. The sequence consists of micro-instructions which can be interpreted as a series of bit fields. A bit field is associated with exactly one particular micro operation. When a field is executed it's value is interpreted to provide the control for this function during that clock period. Descriptor specification: Descriptors can be used separately or application kernels can be chosen from a function library [8]. Plasma program: This is generated through the GNU-C language compiler. The compiler generates an MIPS-I instruction compatible code for the Plasma processor. The programming paradigm allows to find the maximum parallelism found in the application. Exploiting this parallelism allows the high computation rates which are necessary to achieve high performance.

6 Mapping Algorithms and Performance Analysis

This section discusses the mapping of some applications onto the Configurable System-on-Chip architecture. First, the Fast Fourier Transform (FFT) is mapped due to his high degree of important and tight real-time constraints for telecommunication and multimedia systems. Then other typical kernels are mapped and some results becomes introduced.

The Sande and Tukey Fast Fourier Transformation (FFT) algorithm with DIF (Decimation in Frequency) is chosen [9]. It can be more efficiently mapped on the CSoC hardware resources due to a better processing resource utilization. The butterfly calculation is the basic operation of the FFT. An implementation use a DIF-FFT radix-2 butterfly. It is shown in figure 5a). The radix-2 butterfly takes a pair of input data values "X" and "Y" and produces a pair of outputs "X'" and "Y'" where

$$X' = X + Y \ , \ Y' = X - Y\, W \frac{k}{N}. \tag{1}$$

In general the input samples as well as the twiddle factors are complex and can be expressed as

$$X = X_r + jX_i \ \ , \ \ Y = Y_r + jY_i, \tag{2}$$

$$W\frac{k}{N} = e^{-j2\pi k/N} = cos(2\pi k/N) - jsin(2\pi k/N) \tag{3}$$

where the suffix r indicates the real part and the i the imaginary part of the data. In order to execute a 24-bit radix-2 butterfly in parallel, a set of descriptors for the RPCA and the mTC microcode for controlling the heterogeneous processing resources and the configuration manager are necessary. The butterfly operation requires four real multiplications and six real additions/subtractions. To map the complex multiplication, two multiplier and two multiplier/add descriptors as configuration templates are needed. It can be mapped in parallel onto four processing cells of the RPCA. The complex- adder and subtracter are mapped onto the ALU block. The whole structure is shown in figure 5b). The pipelined execution of the radix-2 butterfly takes five clock cycles. The execution is best illustrated using the reservation table. It shows the hardware resource allocation in every clock cycle. Figure 6 shows the reservation table for the first stage of an n-point FFT fragmented into three phases. The *initialize* phase is comprised of static configurations of the processing resources. It initiates the configuration which maps four complex multiplier onto the RPCA by using the CONFIG instruction. In the *configuration* phase, the configuration of the reconfigurable array is accomplished. The *process* phase starts the computation of the n-points. It is controlled by the FTEST instruction. The twiddle-factors $W\frac{k}{N}$ are generated for each stage via the lookup-table and the address generation units.

The example in figure 7 shows the execution of the n-point radix-2 FFT. Each oval in the figure corresponds to the execution of a FFT compute stage (micro-task), while each arrow represents a data stream transfer. The example uses an in-place FFT implementation variant. It allows the results of each FFT butterfly to replace its input. This makes efficient use of the dual ported scratch-pad memories as the transformed data overwrites the input data. The reorder of the output data is simply arranged by reversing the address bits via the scratch-pad memory data sequencer unit (bit-reverse addressing). For demonstration, a 64- point FFT computation, a 16x16 Matrix-Vector-Multiplication (MVM), a 32-tap systolic FIR-filter and 32-tap symmetrical FIR-filter with 12-bit integer coefficients and data and a 32-tap real IIR filter implementation are mapped. Generally larger kernels can be calculated by time-division multiple access (TDMA).

One key to achieve high performance is keeping each functional unit as busy as possible. This goal can be quantified by efficiently mapping applications to the CSoC model including how to partition the kernel structure or large computations. An important aspect is the resource occupancy of the architecture; the percentage of used hardware resources which are involved to process a kernel or an application. The occupancy of the architecture resources for several kernels

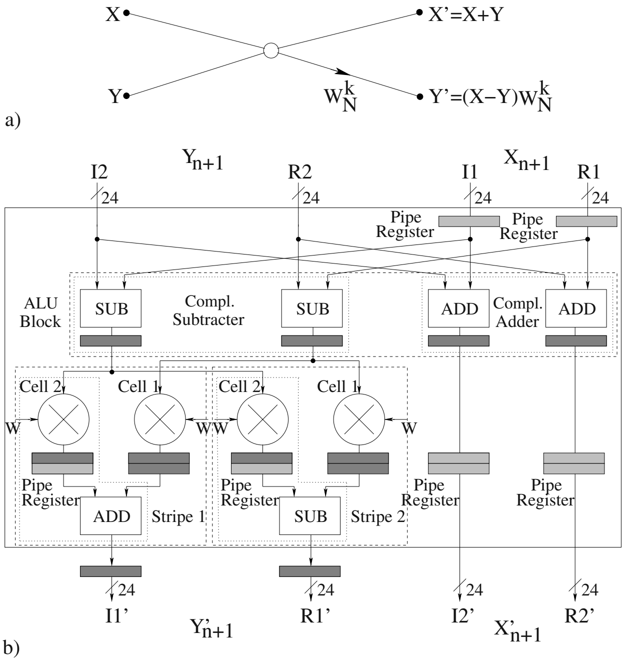
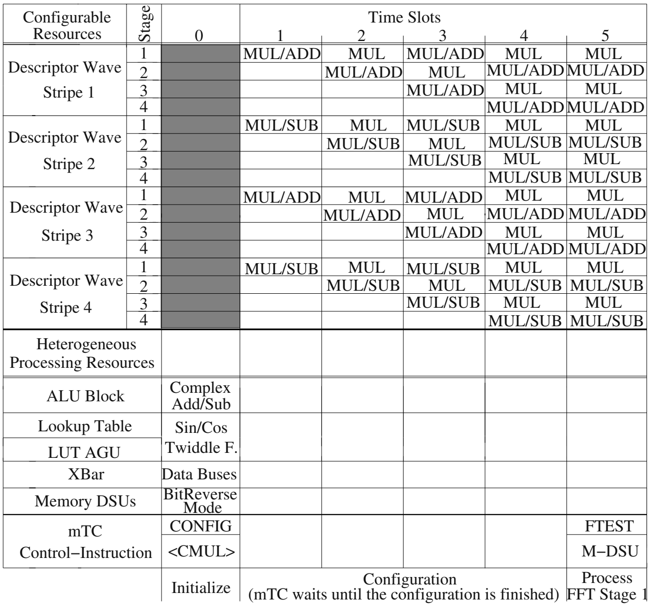


Fig. 5. The butterfly structure for the decimation in frequency radix-2 FFT in a). In b) the mapping of the complex adders onto the ALU block of the heterogeneous processing resources and the complex multiplier which are mapped onto the reconfigurable processing array. The implementation uses additional pipeline registers (Pipe Register) provided by the configuration manager (light hatched) for timing balance.

are shown in figure 8 a). The hierarchically architecture composition allows to map four complex radix-2 butterflies in parallel onto the CSoC with a single cluster. As shown, the complex butterfly can be efficiently implemented due to the mapping of the complex adders to the heterogeneous processing resources and the complex multipliers to the reconfigurable processing cells. The resource unit occupancy of the cluster (Reconfigurable Resources) is 75%. The expensive multipliers in the reconfigurable processing cells are completely used.

The 16x16 MVM kernel is mapped and calculated in parallel with 24-bit precision. The partial data accumulation is done via the adder resources in the ALU block. A 48-bit MVM calculation is not feasible due to limited broadcast bus resources in the cluster [8]. This results in a relative poor resource occupancy as a 24-bit multiplier in the reconfigurable processing cells can not be used. The FIR-filter structures can be mapped directly onto the RPCA. It can be calculated with 48-bit precision. As a result of insufficient adder resources in a reconfigurable processing cell, the 32-tap symmetrical FIR-filter can only be calculated with 24-bit precision. The resource occupancy in a reconfigurable processing cell is poor as a 24-bit multiplier can not be used [8]. The IIR-filter



Configurable Resources	Stage	0	Time Slots 1	2	3	4	5
Descriptor Wave Stripe 1	1	▨	MUL/ADD	MUL	MUL/ADD	MUL	MUL
	2	▨		MUL/ADD	MUL	MUL/ADD	MUL/ADD
	3	▨			MUL/ADD	MUL	MUL
	4	▨				MUL/ADD	MUL/ADD
Descriptor Wave Stripe 2	1	▨	MUL/SUB	MUL	MUL/SUB	MUL	MUL
	2	▨		MUL/SUB	MUL	MUL/SUB	MUL/SUB
	3	▨			MUL/SUB	MUL	MUL
	4	▨				MUL/SUB	MUL/SUB
Descriptor Wave Stripe 3	1	▨	MUL/ADD	MUL	MUL/ADD	MUL	MUL
	2	▨		MUL/ADD	MUL	MUL/ADD	MUL/ADD
	3	▨			MUL/ADD	MUL	MUL
	4	▨				MUL/ADD	MUL/ADD
Descriptor Wave Stripe 4	1	▨	MUL/SUB	MUL	MUL/SUB	MUL	MUL
	2	▨		MUL/SUB	MUL	MUL/SUB	MUL/SUB
	3	▨			MUL/SUB	MUL	MUL
	4	▨				MUL/SUB	MUL/SUB
Heterogeneous Processing Resources							
ALU Block		Complex Add/Sub					
Lookup Table		Sin/Cos					
LUT AGU		Twiddle F.					
XBar		Data Buses					
Memory DSUs		BitReverse Mode					
mTC Control–Instruction		CONFIG <CMUL>					FTEST M–DSU
		Initialize	Configuration (mTC waits until the configuration is finished)				Process FFT Stage 1

Fig. 6. The simplified reservation table of the first FFT stage to process four complex butterflies in parallel. The hardware resources are listed on the left-hand side. First the time slots for the reconfigurable processing array (four parallel stripes) with the pipe stages (Stage), then the heterogeneous processing resources and the mTC control-instructions are illustrated. The hatched area marks the initialization phase. The reconfigurable processing array is configured after the fourth cycle. The Memory Data Sequencer Units (M-DSU) starts after the configuration process (time slot 5) to provide the input data. The heterogeneous and configurable processing resources are statically configured.

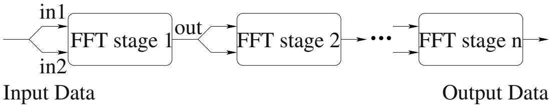
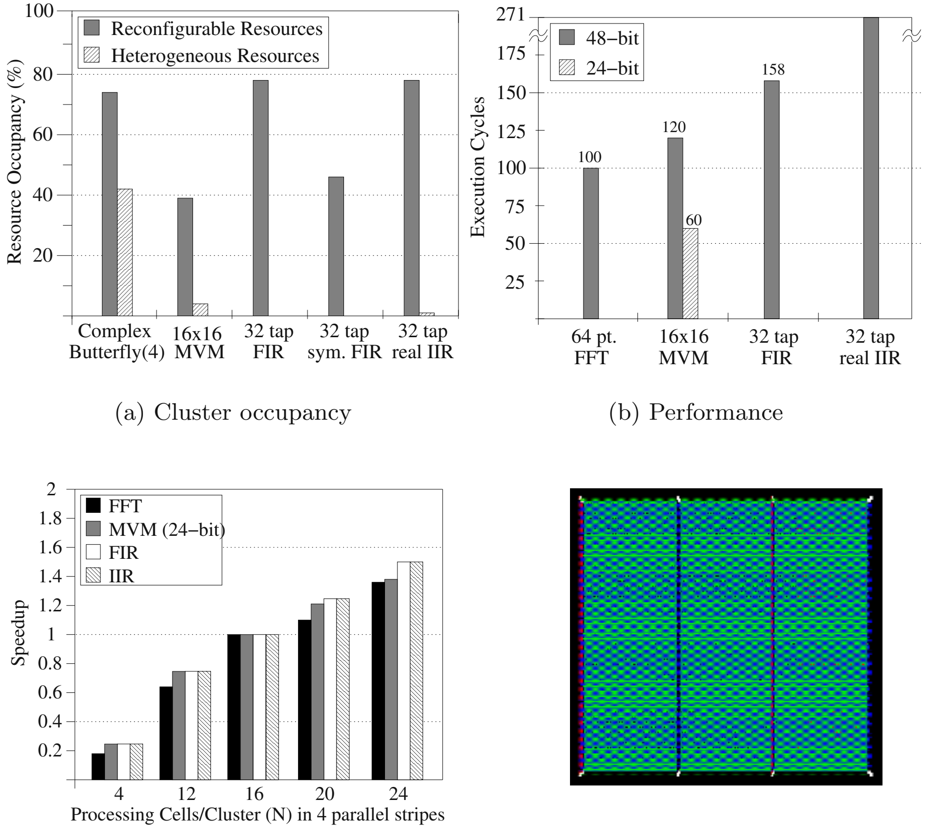


Fig. 7. A n-point FFT (n=max. 1024) is split onto several compute stages (micro-tasks) which are performed sequentially. The compute stages are separated into several parts in the reservation table.

is composed of two parallel 32-tap systolic FIR-filters and an adder from the ALU block. It can only be calculated with full 48-bit precision by using TDMA processing. The lack of wiring resources (broadcast buses in a cluster) limits the resource load in a cluster as the MVM and the 32-tap symmetrical FIR-filter mapping illustrates. The wiring problem cannot always be completely hidden but can be reduced as shown in the case of the mapping of the complex butterfly.



(a) Cluster occupancy

(b) Performance

(c) Speedup

(d) {CSoC layout

Fig. 8. (a) Resource unit occupancy of selected kernels on the CSoC architecture with a single cluster. The dark area shows the cluster utilization while the striped area shows the functional unit utilization of the heterogeneous processing resources. (b) Performance results for the applications using a single cluster. The results do not include the configuration cycles. (c) Speedup results relative to a single cluster with 16 reconfigurable processing cells for selected kernels. Plot (d) shows the CSoC ASIC prototype layout without the pad ring.

Achieved performance results for some of the kernels above are summarized in figure 8 b). It shows the 24- and 48-bit realization of the MVM kernel.

The 48-bit MVM computation needs twice as much clock cycles as the 24-bit realization due to the broadcast bus bottleneck. The other kernels can be calculated with 48-bit precision. The 32-tap FIR-filter can be executed onto the processing array by feeding back the partial data. The 64-point FFT uses four complex butterflies in parallel. The 32-tap IIR-filter needs more then twice as much clock cycles due to the need of TDMA processing to compute the 32-tap FIR-filters in parallel. Speedups with reconfigurable processing cell scaling in four parallel stripes is shown in figure 8 c). The filter kernels have linear speedups to $N = 24$ because they can be directly mapped onto the RPCA. The MVM computation has linear speedup until $N = 16$. The increased number of processing cells gives an performance decrease due to the broadcast bus bottleneck for a stripe. The FFT computation with the parallel implementation of the butterflies shows a resource bottleneck in the reconfigurable processing cells for $N = 4$ and $N = 12$. In this case, it is not possible to map complete complex multipliers in parallel onto the RPCA. In the case of $N = 20$, the broadcast bus bottleneck and a ALU adder bottleneck decrease the performance rapidly due to the absence of adder resources for the butterfly computation. The performance degradation in the $N = 24$ case results from the ALU adder resource bottleneck, the complex multipliers can be mapped perfectly onto four parallel stripes with 24 processing cells. Scaling the number of clusters results in a near-linear speedup for the most kernels unless a resource bottleneck in the heterogeneous resources occurs.

7 Design and Physical Implementation

The CSoC architecture has been implemented using a standard cell design methodology for an UMC 0.18 micron, six metal layer CMOS ($1.8V$) process available through the european joined academic/industry project *EUROPRACTICE*. The architecture was modeled using VHDL as hardware description language. The RPCA and the mTC with the heterogeneous processing resources are simulated separately with the *VHDL System-Simulator (VSS)*. They were then synthesized using appropriate timing constraints with the *Synopsys Design-Compiler (DC)* and mapped using *Cadence Silicon Ensemble*. The first implementation includes a single cluster with 16 processing cells and the mTC unit with the heterogeneous processing resources. The Plasma scalar processor includes a $1k * 32$ program cache. No further caches are implemented. The final layout, shown in figure 8 d), has a total area of $10\,mm^2$. A cluster with 16 processing cells needs $7.1\,mm^2$ silicon area, the Plasma processor core in 0.18 micron CMOS technology approximately $0.4\,mm^2$. It can be clocked up to 210 MHz. After a static timing analysis with *Cadence Pearl*, the CSoC design runs at clock frequencies up to 140MHz.

8 Conclusions and Future Work

A Configurable System-on-Chip for embedded devices has been described in this paper. The CSoC introduces an architecture for telecommunication-, audio-, and video algorithms in order to meet the high performance and flexibility demands. The architecture provides high computational density and flexibility of changing behaviors during run-time. The system consists of two clusters of reconfigurable processing cells, heterogeneous processing resources and a programmable micro Task Controller unit with a small instruction set. The reconfigurable processing array has been designed for data-parallel and computation-intensive tasks. However, the CSoC architecture proposed here is different from many CSoC architectures in an important and fundamental way. It is hierarchically organized and offers a programming model. The architecture allows an efficient mapping of application kernels as the FFT mapping have illustrated and is scalable by either adding more reconfigurable processing cells in a cluster, by increasing the number of clusters or by adding more processing units to the heterogeneous resources. A major advantage of the CSoC architecture approach is to provide forward compatability with other CSoC architecture families. They may offer a different number of clusters or reconfigurable processing cells to provide variable performance requirements. The micro-task program and the descriptor set can be reused. A prototype chip layout with a single cluster has been developed using a UMC 0.18 micron 6-layer CMOS process.

Apart from tuning and evaluating the CSoC architecture for additional applications, there are several directions for future work. One key challenge is to extended the mTC instruction in order to enlarge the flexibility. Another challenge is to create an automatic partitioning and mapping tool to assist the user. The programmable micro Task Controller and the reconfigurable processing array with the descriptors as configuration templates can be a solid base for such an intention.

References

1. R. Berezdivin, R. Breinig, R. Topp, “Next-generation wireless communication concepts and technologies”, IEEE Communication Magazine, vol. 40, no. 3, pp. 108-117, Mar. 2002
2. K. Lewis, “Information Appliances, “Gadget Netopia”, IEEE Computer., vol. 31, pp. 59-66, Jan. 1998
3. K. Diefendorff and P. Dubey, “How Multimedia Workloads Will Change Processor Design”, IEEE Computer, 30(9) : 43-45, Sept. 1997
4. T. M. Conte, P. K. Dubey, M. D. Jennings, R. B. Lee, A. Peleg, S. Rathnam, M. S. Schlansker, P. Song, A. Wolfe, “Challenges to Combining General-Purpose and Multimedia Processors”, IEEE Computer, pp. 33-37, Dec. 1997
5. M. Wan, H. Zhang, V. George , M. Benes, A. Abnous, V. Prabhu, J. Rabaey, ”Design methodology of a low-energy reconfigurable single-chip DSP system”, Journal of VLSI Signal Processing, vol.28, no.1-2, pp.47-61, May-Jun. 2001
6. B. Salefski, L. Caglar, ”Re-Configurable Computing in Wireless”, 38th Design Automation Conference, Las Vegas, Nevade, USA, Jun. 2001
7. H. Singh, M. H. Lee, G. Lu, F. J. Kurdahi, N. Bagherzadeh, E. M. C. Filho, ”MorphoSys: An Integrated Reconfigurable System for Data-Parallel and Computation-Intensive Applications”, IEEE Transactions on Computers 49(5): 465-481, May 2000
8. S. Wallner, ”A Reconfigurable Multi-threaded Architecture Model”, Eighth Asia-Pacific Computer Systems Architecture Conference (ACSAC 2003), Fukushima, Japan, Springer LNCS 2823, pp. 193-207, Sep. 23-26 2003
9. A. V. Oppenheim, R. W. Schafer, ”Discrete-Time Signal Processing”, Englewood Cliffs, Prentice-Hall, 1989

An Auto-adaptative Reconfigurable Architecture for the Control

Nicolas Ventroux, Stéphane Chevobbe, Fréderic Blanc, and Thierry Collette

CEA-List DRT/DTSI/SARC
Image and Embedded Computers Laboratory
F-91191 Gif-Sur-Yvette - FRANCE
phone: (33) 1-69-08-66-37
firstname.surname@cea.fr

Abstract. Previous works have shown that reconfigurable architectures are particularly well-adapted for implementing regular processing applications. Nevertheless, they are inefficient for designing complex control systems. In order to solve this drawback, microprocessors are jointly used with reconfigurable devices. However, only regular, modular and reconfigurable architectures can easily take into account constant technology improvements, since they are based on the repetition of small units. This paper focuses on the self-adaptative features of a new reconfigurable architecture dedicated to the control from the application to the computation level. This reconfigurable device can itself adapt its resources to the application at run-time, and can exploit a high level of parallelism into an architecture called *RAMPASS*.

Keywords: dynamic reconfiguration, adaptative reconfigurable architecture, control parallelism

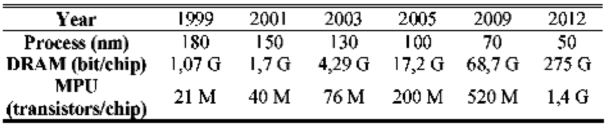
1 Introduction

The silicon area of reconfigurable devices are filled with a large number of computing primitives, interconnected via a configurable network. The functionality of each element can be programmed as well as the interconnect pattern. These regular and modular structures are adapted to exploit future microelectronic technology improvements. In fact, semiconductor road maps [1] indicate that integration density of regular structures (like memories) increases faster than irregular ones (Tab. 1). In this introduction, existing reconfigurable architectures as well as solutions to control these structures, are first presented. This permits us to highlight the interests of our architecture dedicated to the control, which is then depicted in details.

A reconfigurable circuit can adapt its features, completely or partially, to applications during a process called reconfiguration. These reconfigurations are statically or dynamically managed by hardware mechanisms [2]. These architectures can efficiently perform hardware computations, while retaining much of

P.-C. Yew and J. Xue (Eds.): ACSAC 2004, LNCS 3189, pp. 72–87, 2004.

Table 1. Integration density for future VLSI devices



Year	1999	2001	2003	2005	2009	2012
Process (nm)	180	150	130	100	70	50
DRAM (bit/chip)	1,07 G	1,7 G	4,29 G	17,2 G	68,7 G	275 G
MPU (transistors/chip)	21 M	40 M	76 M	200 M	520 M	1,4 G

the flexibility of a software solution [3]. Their resources can be arranged to implement specific and heterogeneous applications. Three kinds of reconfiguration level can be distinguished :

- **gate level**: FPGA (Field Programmable Gate Array) are the most well-known and used gate-level reconfigurable architectures [4,5]. These devices merge three kinds of resources: the first one is an interconnection network, the second one is a set of processing blocks (LUT, registers, etc.) and the third one regroups I/O blocks. The reconfiguration process consists in using the interconnection network to connect different reconfigurable processing elements. Furthermore, each LUT is configured to perform any logical operations on its inputs. These devices can exploit bit-level parallelism.
- **operator level**: the reconfiguration takes place at the interconnection and the operator levels (PipeRench[6], DREAM [7], MorphoSys [8], REMARC [9], etc.). The main difference concerns the reconfiguration granularity, which is at the word level. The use of coarse-grain reconfigurable operators provides significant savings in time and area for word-based applications. They preserve a high level of flexibility in spite of the limitations imposed by the use of coarse-grain operators for better performances, which do not allow bit-level parallelism.
- **functional level**: these architectures have been developed in order to implement intensive arithmetic computing applications (RaPiD [10], DART [11], Systolic Ring [12], etc.). These reconfigurable architectures are reconfigured in modifying the way their functional units are interconnected. The low reconfiguration data volume of these architectures makes it easier to implement dynamic reconfigurations and allows the definition of simple execution models.

These architectures can own different levels of physical granularity, and whatever the reconfiguration grain is, partial reconfigurations are possible, allowing the virtualization of their resources. Thus for instance, to increase performances (area, consumption, speed, etc.), an application based on arithmetic operators is optimally implemented on word-level reconfigurable architectures.

Besides, according to Amdahl's law [13], an application is always composed of regular and irregular processings. It is always possible to reduce and optimize the regular parts of an application in increasing the parallelism, but irregular code is irreductible. Moreover, it is difficult to map these irregular parts on reconfigurable architectures. Therefore, most reconfigurable systems need to be coupled with an external controller especially for irregular processing or dynamic context switching. Performances are directly dependent on the position and the level of this coupling. Presently, four possibilities can be exploited by designers:

- **microprocessor**: this solution is often chosen when reconfigurable units are used as coprocessors in a SoC (System on Chip) framework. The microprocessor can both execute its own processes (and irregular codes) and configure its reconfigurable resources (PACT XPP-/Leon [14], PipeRench [15], MorphoSys [8], DREAM [7], etc.). These systems can execute critical processings on the microprocessor, while other concurrent processes can be executed on reconfigurable units. However, in order to only configure and execute irregular code, this solution may be considered as too expensive in terms of area and energy consumption, and would most likely be the bottelneck due to off-chip communication overheads in synchronization and instruction bandwidth.

- **processor core**: this approach is completely different since the processor is mainly used as a reconfigurable unit controller. A processor is inserted near reconfigurable resources to configure them and to execute irregular processes. Performances can also be increased by exploiting the control parallelism thanks to tight coupling (Matrix [16], Chimaera [17], NAPA [18], etc.). Marginal improvements are often noticed compared to a general-purpose microprocessor but these solutions give an adapted answer for controlling reconfigurable devices.

- **microsequencer**: these control elements are only used to process irregular processing or to configure resources. They can be found in the RaPiD architecture, for instance, [10] as a smaller programmed control with a short instruction set. Furthermore, the GARP architecture uses a processor in order to only load and execute array configurations [19]. A microsequencer is an optimal solution in terms of area and speed. Its features do not allow itself to be considered as a coprocessor like the other solutions, but this approach is however best fitted for specifically controlling reconfigurable units. Nevertheless, control parallelisms can be exploited with difficulty.

- **FPGA**: this last solution consists in converting the control into a set of state machines, which could then be mapped to an FPGA. This approach can take advantage of traditional synthesis techniques for optimizing control. However, FPGA are not optimized for implementing FSM (Finite State Machines) because whole graphs of the application must be implemented even if non-deterministic processes occur. Indeed, these devices can hardly manage dynamic reconfigurations at the state-level.

Reconfigurable devices are often used with a processor for non-deterministic processes. To minimize control and configuration overheads, the best solution consists in tightly coupling a processor core with the reconfigurable architecture [20]. However, designing for such systems is similar to a HW/SW co-design problem. In addition, the use of reconfigurable devices can be better adapted to deep sub-microelectronic technological improvements. Nonetheless, the controller needs other physical implementation features rather than operators, and FPGA can not always be an optimal solution for computation. Indeed, the control handles small data and requires global communications to control all the processing elements, whereas the computation processes large data and uses local communications between operators.

To deal with control for reconfigurable architectures, we have developed the *RAMPASS* architecture (Reconfigurable and Advanced Multi-Processing Architecture for future Silicon Systems) [21]. It is composed of two reconfigurable resources. The first one is suitable for computation purposes but is not a topic of interest for this paper. The second part of our architecture is dedicated to control processes. It is a self-reconfigurable and asynchronous architecture, which supports SIMD (Single Instruction Multiple Data), MIMD (Multiple Instruction Multiple Data) and multi-threading processes.

This paper presents the mechanisms used to auto-adapt resource allocations to the application in the control part of *RAMPASS*. The paper is structured as follows: section 2 outlines a functional description of *RAMPASS*. Section 3 presents a detailed functional description of the part of *RAMPASS* dedicated to the control. This presentation focuses on some concepts presented in [21]. Then, section 4 depicts auto-adaptative reconfiguration mechanisms of this control part. Finally, section 5 presents the development flow, some results and deals with the SystemC model of our architecture.

2 Functional Description of RAMPASS

In this section, the global functionality of *RAMPASS* is described. It is composed of two main reconfigurable parts (Fig. 1):

- One dedicated to the control of applications (*RAC*: Reconfigurable Adapted to the Control);
- One dedicated to the computation (*RAO*: Reconfigurable Adapted to Operators).

Even if the *RAC* is a part of *RAMPASS*, it can be dissociated to be integrated with other different architectures with any computational grain. Each computation block can be either a general-purpose processor or a functional unit. The *RAC* is a generic control architecture and is the main interest of this paper. In this article, the *RAO* can be considered as a computational device adapted to the application, with a specific interface in order to communicate with the *RAC*. This interface must support instructions from the *RAC* and return one-bit flags according to its processes.

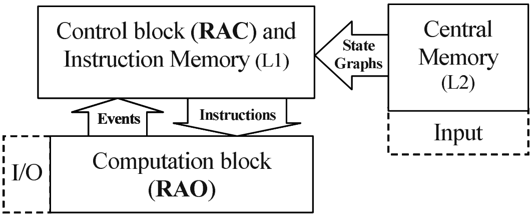


Fig. 1. Organization of RAMPASS

2.1 Overview

From a C description, any application can be translated as a *CDFG* (Control Data Flow Graph), which is a *CFG* (Control Flow Graph) with the instructions of the basic blocks expressed as a *DFG* (Data Flow Graph). Thus, their partition is easily conceivable [22,23].

A *CFG* or a *State Graph* (*SG*) represents the control relationships between the set of basic blocks. Each basic block contains a set of deterministic instructions, called *actions*. Thus, every state in a SG is linked to an action. Besides, every arc in a SG either connects a state to a transition, or a transition to a state. A SG executes by firing transitions. When a transition fires, one token is removed from each input state of the transition and one token is added to each output state of the transition. These transistions determine the appropriate control edge to follow. On the other hand, a *DFG* represents the overall corresponding method compiled onto hardware.

Consequently, whatever the application is, it can be composed of two different parts (Fig. 2). The first one computes operations (*DFG*) and the second one schedules these executions on a limited amount of processing resources (*CFG*).

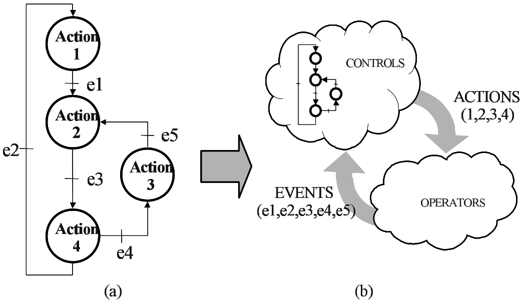


Fig. 2. Partitioning of an application (a) in Control/Computation (b)

The first block of our architecture can physically store any application described as a *CFG*. States drive the computation elements in the *RAO*, and events coming from the *RAO* validate transitions in the SG. Moreover, self-routing mechanisms have been introduced in the *RAC* block to simplify SG mapping. The *RAC* can auto-implement a SG according to its free resources. The *RAC* controls connections between cells and manages its resources. All these mechanisms will be discussed in future sections.

2.2 Mapping and Running an Application with RAMPASS

In this part, the configuration and the execution of an application in *RAMPASS* are described. Applications are stored in an external memory. As soon as the SG begins to be loaded in the *RAC*, its execution begins. In fact, the configuration and the execution are simultaneously performed. Contrary to microprocessor, this has the advantage of never blocking the execution of applications, since the following executed actions are always mapped in the *RAC*.

The reconfiguration of the *RAC* is self-managed and depends on the application progress. This concept is called auto-adaptative. The *RAC Net* has a limited number of cells, which must be dynamically used in order to map larger applications. Indeed, due to a lack of resources, whole SGs can not always be mapped in the *RAC*. Dynamic reconfiguration has been introduced to increase the virtual size of the architecture. In our approach, no pre-divided contexts are required. Sub-blocks implemented in the *RAC Net* are continuously updated without any user help. Figure 3 shows a sub-graph of a 7-state application implemented at run-time in a 3-cell *RAC* according to the position of the token.

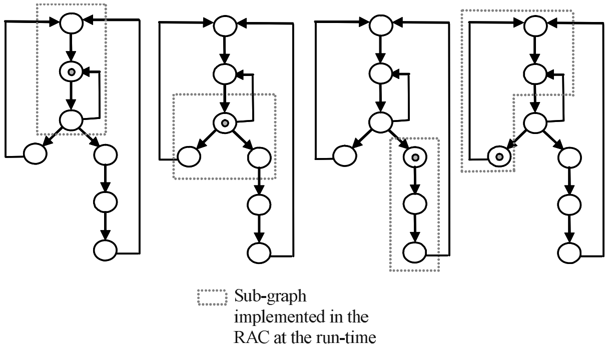


Fig. 3. Evolution of an implemented SG in the RAC Net

Each time a token is received in a cell of a SG implemented in the *RAC*, its associated instructions are sent to the *RAO*. When the *RAO* has finished its processes, it returns an event to the cell. This event corresponds to an edge in the SG mapped in the *RAC*. These transitions permit the propagation of tokens in SGs. Besides, each block has its synchronization mechanisms. In this globally asynchronous architecture, blocks are synchronized by 2-phase protocols [24].

It is possible to execute concurrently any parallel branches of a SG, or any independant SGs in the *RAC*. This ensures *SIMD*, *MIMD*, and multi-threading control parallelisms. Besides, semaphore and mutex can be directly mapped inside the *RAC* in order to manage shared resources or synchronization between SGs. Even if SGs are implemented cell by cell, their instantiations are concurrent.

3 Functional Description of the Control Block: The RAC

As previously mentioned, the *RAC* is a reconfigurable block dedicated to the control of an application. It is composed of five units (Fig. 4). The *CPL* (*Configuration Protocol Layer*), the *CAM* (*Content Addressable Memory*) and the *LeafFinder* are used to configure the *RAC Net* and to load the *Instruction Memory*.

3.1 Overview

The *RAC Net* can support physical implementation of SGs. When a cell is configured in the *RAC Net*, its associated instructions are stored in the *Instruction Memory* as well as the address of its description in the *CAM*. Descriptions of cells are placed in a central memory and each description contains the instruction of the associated cell and the configuration of cells, which must be connected (daughter cells). In order to extend SGs in the *RAC Net*, the last cells of SGs, which are called *leaf cells*, are identified in the *LeafFinder*. These cells allow the extension of SGs. When a leaf cell is detected, a signal is sent to the *CAM* and the description of this cell is read in the central memory. From this description,

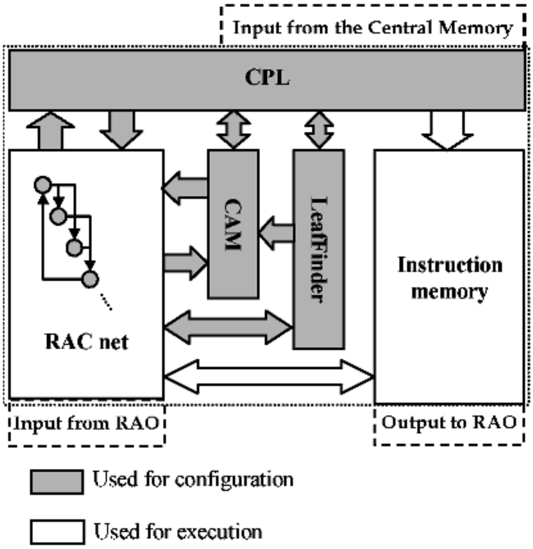


Fig. 4. The RAC block

the daughter cells of this leaf cell are configured and links are established between the cells in the *RAC Net*. The *CAM* can also find a cell mapped in the *RAC Net* thanks to its address. This is necessary if loop kernels try to connect already mapped cells. Finally, the propagation of tokens through SGs, thanks to events from the *RAO*, schedule the execution of instructions stored in the *Instruction Memory*. In the next section, the details of each block are given.

3.2 Blocks Description

RAC Net, this element is composed of cells and interconnect components. SGs are physically implemented thanks to these resources. One state of a SG is implemented by one cell. Each cell directly drives instructions, which are sent to the *RAO*. The *RAC Net* is dynamically reconfigurable. Its resources can be released or used at the run-time according to the execution of the application. Moreover, configuration and execution of SGs are fully concurrent. *RAC Net* owns primitives to ensure the auto-routing and the managing of its resources (cf §4.1). The *RAC Net* is composed of three one-hot asynchronous FSMs (5 ,8 and 2 states) to ensure the propagation of tokens, its dynamical destruction and the creation of connections. It represents about one thousand transistors in ST 0.18μm technology.

Instruction memory, the *Instruction Memory* contains the instructions, which are sent by the *RAC Net* to the *RAO* when tokens run through SGs. An instruction can eventually be either configurations or context addresses. As shown in figure 5, the split instruction bus allows the support of *EPIC* (Explicitly Parallel Instruction Computing) and the different kinds of parallelism introduced in the first section. Each column is reserved for a computation block in the *RAO*. For instance, the instructions A and B could be sent together to different computational blocks mapped in the *RAO* without creating conflicts, whereas the instruction C would be sent alone. A bit of selection is also used to minimize energy consumption by disabling unused blocks.

Furthermore, each line is separately driven by a state, e.g. each cell of the *RAC Net* is dedicated to the management of one line of this memory. This memory does not require address decoding since its access is directly done through its word lines. We call this kind of memory a *word-line memory*.

CPL, this unit manages SG implementation in the *RAC Net*. It sends all the useful information to connect cells, which can auto-route themselves. It drives either a new connection if the next state is not mapped in the *RAC net*, or a connection between two states already mapped. It also sends primitives to release resources when the *RAC Net* is full.

CAM, this memory links each cell of the *RAC Net* used to map a state of a SG, with its address in the external memory. Again, it can be driven directly

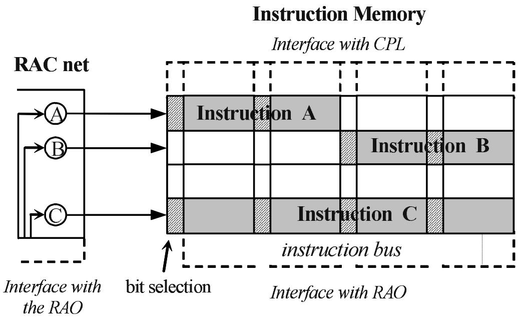


Fig. 5. Relation RAC Net/Instruction Memory

through its word lines. It is used by the *CPL* to check if a cell is already mapped in the *RAC Net*. The *CAM* can select a cell in the *RAC Net* when its address is presented by the *CPL* at its input. Besides, the *CAM* contains the size of cell descriptions to optimize the bandwidth with the central memory.

LeafFinder, this word-line memory identifies all the leaf cells. Leaf cells are in a semi-mapped state which does not yet have an associated instruction. The research is done by a logic ring, which runs each time a leaf cell appears.

4 Auto-adaptative Reconfiguration Control

The first part of this section deals with the creation of connections between cells and their configuration. A cell, which takes part in a SG, must be configured in a special state corresponding to its function in the SG. Finally, the second part focuses on the release of already used cells.

4.1 Graph Creation and Configuration

New connection. To realize a new connection e.g. a connection with a free cell, the *CPL* sends a primitive called *connection*. This carries out automatically a connection between an existing cell (the source cell), which is driven by the *LeafFinder*, and a new cell (the target cell), which is a free cell chosen in the neighborhood of the source cell. Thus, each daughter in the neighborhood of the source cell are successively tested until a free cell is found. The *RAC Net* and its network can self-manage these connections. In fact, carrying out a connection consists of validating existing physical connections between both cells. Finally, the path between the two cells can be considered as auto-routed in the *RAC Net*.

Connection between two existing cells. When the *RAC* finds the two cells, which must be connected, two primitives called *preparation* and *search* are successively sent by the *CPL* to the *RAC Net*. The first one initializes the research process and the second one executes it. The source cell is driven by the *LeafFinder* via the signal *start* and the target cell by the *CAM* via the signal *finish*. According to the application, the network of the RAC Net can be either fully or partially interconnected. Indeed, the interconnection network area is a function of the square of the number of cells in the RAC Net. Thus, a fully connected network should be used only in highly irregular computing application.

If the network is fully interconnected, the connection is simply done by the interconnect, which receives both the signals *start* and *finish*. On the other hand, if cells are partially interconnected, handshaking mechanisms allow the source cells to find the target. Two signals called *find* and *found* link each cells together (Fig. 6). On the reception of the signal *search*, the source cell sends a *find* signal to its daughters. The free cell receiving this signal sends it again to its daughters (this signal can be received only one time). So, the signal *find* spreads through free cells until it reaches the target cell. Then this cell sends back the signal *found* via the same path to the source cell. Finally, the path is validated and a hardware connection is established between the two cells. The intermediate and free cells, which take part in the connection, are in a special mode named *bypass*.

Configuration. The dynamic management of cells is done by a signal called *accessibility*. This signal links every cell of a SG when a connection is done. Each cell owns an *Up Accessibility* (*UA*) (from its mother cells) and a *Down Accessibility* (*DA*) (distributed to its daughter cells). At the time of a new connection, a cell receives the *UA* from its mother cells and stores its configuration coming from the *CPL*. In the case of multiple convergences (details on SG topologies have been presented in [21]), it receives the *UA* as soon as the first connection is established. Then, a configured cell is ready to receive and to give a token. After its configuration, the cell transmits its *accessibility* to its daughters.

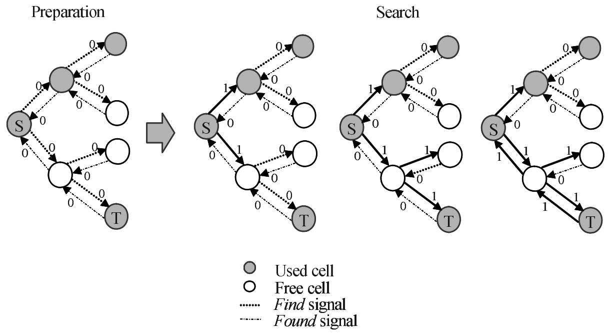


Fig. 6. Connection between the source cell (S) and its target cell (T)

When the connection has succeeded, the *RAC Net* notifies the *CPL*. Consequently, the *CPL* updates the *CAM* with the address of the new mapped state, the *LeafFinder* defines the new cell as a leaf cell, and the *Instruction Memory* stored the correct instructions.

When a connection fails, the *RAC Net* indicates an error to the *CPL*. The *CPL* deallocates resources in the *RAC Net* and searches the next leaf cell with the *LeafFinder*. These two operations are repeated until a connection succeeds. Release mechanisms are detailed in the next paragraph.

4.2 Graph Release

A cell stops to deliver its *accessibility* when it no more receives an *UA* and does not own a token. When a cell loses its accessibility, all the daughter cells are successively free and can be used for other SG implementations. In order to prevent the release of frequently used cells, which may happen in loop kernels, a configuration signal called *stop point* can be used.

Due to resource limitations, a connection attempt may fail. For this reason, a complete error management system has been developed. It is composed of three primitives, which can release more or less cells. The appearing frequency of connection errors is evaluated by the *CPL*. When predefined thresholds are reached, adapted primitives are sent to the *RAC Net* to free unused resources. The first one is called *test acces*. It can free a cell in a *stop point* mode (Fig. 7). Every cell between two *stop point* cells are free. Indeed, a *stop point* cell is free on a rising edge of the *test acces* signal when it receives the *accessibility* from its mothers.

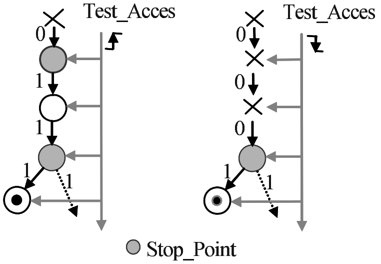


Fig. 7. Releasing of cells with test access

The second release primitive is named *reset stop point*. It can force the liberation of any *stop point* cells when they do not have any token. This mode keeps cells implied in the implementation of loop kernels and reduces the release. In some critical cases (when resources are very limited), it can become an idle state.

Finally, the last primitive called *reset idle state* guarantees no idle state in the *RAC*. This is done by freeing all the cells, which do not own a token. This solution is of course the more efficient but is very expensive in time and energy consumption. It must only be used in case of repeated desallocation errors.

No heuristics decide how many cells must be reclaimed or loaded. This is done automatically even if the desallocation is not optimal. That is why *stop point* cells must be adequately placed in SGs to limit releases.

Non-deterministic algorithms need to make decisions to follow their processes. This can be translated as OR divergences, e.g. events determine which branch will be followed by firing transitions. To prevent speculative construction and to configure too many unemployed cells, the construction of SGs is blocked until correct decisions are taken. This does not slow the execution of the application since the *RAC Net* contains always the next processes. Moreover, we consider that execution is slower than reconfiguration, and that an optimal computation time is about $3ns$. Indeed, we estimate the reconfiguration time of a cell equals to $7.5ns$ and the minimum time between two successive instructions for a fully interconnected network of $3ns + 1.5ns$, where $1.5ns$ is the interconnect propagation time for a 32-cell *RAC*.

5 Implementation and Performance Estimation

An architecture can not be exploited without a development flow. For this reason, a development flow is currently a major research concern of our laboratory (Fig. 8). From a description of the application in C-language, an intermediate representation can be obtained by a front-end like SUIF [22,23]. Then, a parallelism exploration from the *CDFG* must be done to assign tasks to the multiple computing resources of the *RAO*. This parallelism exploration under constraints increases performances and minimizes the energy consumption and the memory

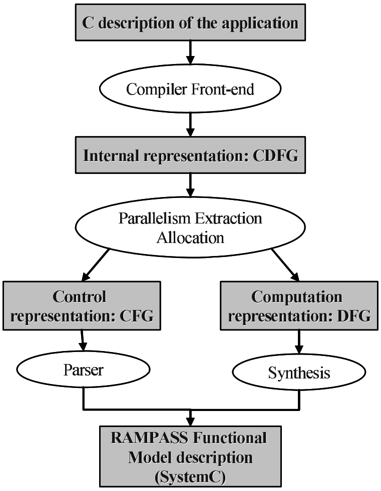


Fig. 8. RAMPASS Development Flow Graph

bandwidth. The allocation of multiple resources in the *RAO* can also increase the level of parallelism. From this optimized *CDFG*, *DFGs* must be extracted in order to be executed on *RAO* resources. Each *DFG* is then translated into *RAO* configurations, thanks to behavioral synthesis scheme. This function is currently under development through the *OSGAR* project, which consists in designing a general-purpose synthesizer for any reconfigurable architectures. This *RNTL* project under the ward of the French research ministry, associates TNI-Valiosys, the Occidental Brittany University and the R2D2 team of the IRISA. On the other hand, a parser used to translate a *CFG* into the *RAMPASS* description language, has been successfully developed.

Besides, a functional model of the *RAC* block has been designed with SystemC. Our functional-level description of the *RAC* is a CABA (Cycle Accurate and Bit Accurate) hardware model. It permits the change of the size and the features of the *RAC Net* and allows the evaluation of its energy consumption. The characteristics of this description language easily allows hardware descriptions, it has the flexibility of the C++ language and brings all the primitives for the modelization of hardware architectures [25,26].

A lot of different programming structures have been implemented in the *RAC* block, e.g. exclusion mechanisms, AND convergence and divergence, synchronizations between separated graphs, etc. Moreover, an application of video processing (spinal search algorithm for motion estimation [27]) has been mapped (Fig. 9). The latency overhead is insignificant without reconfiguration when the

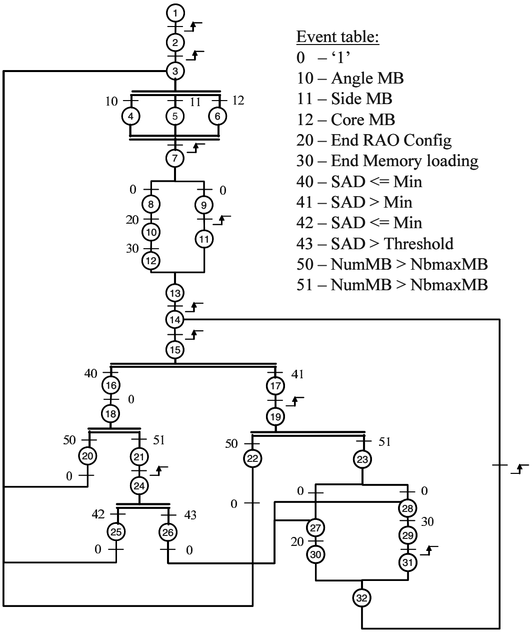


Fig. 9. Motion estimation graph

RAC owns 32 cells, or with a 15-cell *RAC* when the whole main loop kernel can be implemented (0.01%), even if we cannot predict reconfigurations. Finally with a 7-cell *RAC* (the minimal required for this application), the overhead raises only 10% in spite of multiple reconfigurations, since the implementation of the SG must be continuously updated.

Besides, hardware simulations have shown the benefits of release primitives. Indeed, the more cells are released, the more the energy consumption increases since they will have to be re-generated, especially in case of loops. Simulations have shown that these releases are done only when necessary.

Some SG structures implemented in the *RAC Net* need an imperative number of cells. This constraints the minimal number of cells to prevent dead-locks. For instance, a multiple AND divergence has to be entirely mapped before the token is transmitted. Consequently, an 8-state AND divergence needs at least nine cells to work. Dynamic reconfiguration ensures the progress of SGs but can not prevent dead-locks if complex structures need more cells than available inside the *RAC Net*. On the contrary, the user can map a linear SG of thousands of cells with only two free cells.

6 Conclusion and Future Work

New paradigm of dynamically self-reconfigurable architecture has been proposed in this paper. The part depicted is dedicated to the control and can physically implement control graphs of applications. This architecture brings a novel approach for controlling reconfigurable resources. It can answer future technology improvements, allow a high level of parallelism and keep a constant execution flow, even for non-predictible processing.

Our hardware simulation model has successfully validated static and dynamic reconfiguration paradigms. According to these results, further works will be performed. To evaluate performances of RAMPASS, a synthesized model and a prototype of the *RAC* block is currently designed in a ST 0.18μm technology.

Moreover, the coupling between the *RAC* and other reconfigurable architectures (DART, Systolic Ring, etc.) will be studied. The aim of these further collaborations consists in demonstrating the high aptitudes of the *RAC* to adapt itself to different computation architectures.

Acknowledgements. We thank Dominique Lavenier (IRISA, France), Laurent Letellier and Raphaël David (CEA, France) for helpful discussions and comments on this work.

References

1. Semiconductor Industry Association. International Technology Roadmap for Semiconductors. Technical report, 2003.
2. K. Compton and S. Hauck. Reconfigurable Computing: A Survey of Systems and Software. *ACM Computing Surveys*, 34(2):171–210, June 2002.

3. R. Hartenstein. A Decade of Reconfigurable Computing: a Visionary Retrospective. In *IEEE Design Automation and Test in Europe (DATE)*, Munich, Germany, March 2001.
4. Xilinx, http://www.xilinx.com.
5. Altera, http://www.altera.com.
6. S.C. Goldstein, H. Schmit, M. Budiu, S. Cadambi, M. Moe, and R.R. Taylor. PipeRench: A Reconfigurable Architecture and Compiler. *Computer: Innovative Technology for Computer Profesionals*, 33(4):70–77, April 2000.
7. J. Becker, M. Glesner, A. Alsolaim, and J. Starzyk. Fast Communication Mechanisms in Coarse-grained Dynamically Reconfigurable Array Architectures. In *Workshop on Engineering of Reconfigurable Hardware/Software Objects (ENREGLE)*, Las Vegas, USA, June 2000.
8. H. Singh, M.-H. Lee, G. Lu, F.J. Kurdahi, N. Bagherzadeh, and E.M. Chaves Filho. MorphoSys: An Integrated Reconfigurable System for Data-Parallel and Computation-Intensive Applications. *IEEE Trans. on Computers*, Vol.49, No.5:465–481, May 2000.
9. T. Miyamori and K. Olukotun. REMARC: Reconfigurable Multimedia Array Coprocessor. In *ACM/SIGDA Field Programmable Gate Array (FPGA)*, Monterey, USA, February 1998.
10. D. Cronquist. Architecture Design of Reconfigurable Pipelined Datapaths. In *Advanced Research in VLSI (ARVLSI)*, Atlanta, USA, March 1999.
11. R. David, S. Pillement, and O. Sentieys. *Low-Power Electronics Design*, chapter 20: Low-Power Reconfigurable Processors. CRC press edited by C. Piguet, April 2004.
12. G. Sassateli, L. Torres, P. Benoit, T. Gil, G. Cambon, and J. Galy. Highly Scalable Dynamically Reconfigurable Systolic Ring-Architecture for DSP applications. In *IEEE Design Automation and Test in Europe (DATE)*, Paris, France, March 2002.
13. G.M. Amdahl. Validity of the Single-Processor Approach to Achieving Large Scale Computing Capabilities. In *AFIPS Conference Proceedings vol.30*, Atlantic City, USA, April 1967.
14. J. Becker and M. Vorbach. Architecture, Memory and Interface Technology Integration of an Industrial/Academic Configurable System-on-Chip (CSoC). In *IEEE Computer Society Annual Workshop on VLSI (WVLSI)*, Florida, USA, February 2003.
15. Y. Chou, P. Pillai, H. Schmit, and J.P. Shen. PipeRench Implementation of the Instruction Path Coprocessor. In *Symposium on Microarchitecture (MICRO-33)*, Monterey, USA, December 2000.
16. B. Mei, S. Vernalde, D. Verkest, and R. Lauwereins. Design Methodology for a Tightly Coupled VLIW/Reconfigurable Matrix Architecture: A Case Study. In *Design Automation and Test in Europe (DATE)*, Paris, France, February 2004.
17. Z. Ye, P. Banerjee, S. Hauck, and A. Moshovos. CHIMAERA: A High-Performance Architecture with a Tightly-Coupled RFU. In the *27th Annual International Symposium on Computer Architecture (ISCA)*, Vancouver, Canada, June 2000.
18. C. Rupp, M. Landguth, T. Garverick, E. Gomersall, H. Holt, J. Arnold, and M. Gokhale. The NAPA Adaptive Processing Architecture. In *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, Napa Valley, USA, April 1998.
19. J.R. Hauser and J. Wawrzynek. GARP: A MIPS Processor with a Reconfigurable Coprocessor. In *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, Napa Valley, USA, April 1997.

20. D. Rizzo and O. Colavin. A Video Compression Case Study on a Reconfigurable VLIW Architecture. In *Design Automation and Test in Europe (DATE)*, Paris, France, March 2002.
21. S. Chevobbe, N. Ventroux, F. Blanc, and T. Collette. RAMPASS: Reconfigurable and Advanced Multi-Processing Architecture for future Silicon System. In *3rd International Workshop on Systems, Architectures, Modeling and Simulation (SAMOS)*, Samos, Greece, July 2003.
22. G. Aigner, A. Diwan, D.L. Heine, M.S. Lam, D.L. Moore, B.R. Murphy, and C. Sapuntzakis. The Basic SUIF Programming Guide. Technical report, Computer Systems Laboratory, Stanford University, USA, August 2000.
23. M.D. Smith and G. Holloway. An Introduction to Machine SUIF and its Portable Libraries for Analysis and Optimization. Technical report, Division of Engineering and Applied Sciences, Harvard University, USA, July 2002.
24. I.E. Sutherland. Micropipelines. *Communications of the ACM*, 32(6):720–738, 1989.
25. J. Gerlach and W. Rosenstiel. System level design using the SystemC modeling platform. In *the 3rd Workshop on System Design Automation (SDA)*, Rathen, Germany, 2000.
26. S. Swan. An Introduction to System Level Modeling in SystemC 2.0. Technical report, Cadence Design Systems, Inc., May 2001.
27. T. Zahariadis and D. Kalivas. A Spiral Search Algorithm for Fast Estimation of Block Motion Vectors. In *the 8th European Signal Processing Conference (EUSIPCO)*, Trieste, Italy, September 1996.

Enhancing the Memory Performance of Embedded Systems with the Flexible Sequential and Random Access Memory

Ying Chen, Karthik Ranganathan, Vasudev V. Pai, David J. Lilja, and Kia Bazargan

Department of Electrical and Computer Engineering
University of Minnesota
200 Union St. S.E., Minneapolis, MN 55455, USA
{wildfire, kar, pvasudev, lilja, kia}@ece.umn.edu

Abstract. The on-chip memory performance of embedded systems directly affects the system designers' decision about how to allocate expensive silicon area. We investigate a novel memory architecture, *flexible sequential and random access memory* (FSRAM), for embedded systems. To realize sequential accesses, small "links" are added to each row in the RAM array to point to the next row to be prefetched. The potential cache pollution is ameliorated by a small *sequential access buffer* (SAB). To evaluate the architecture-level performance of FSRAM, we run the Mediabench benchmark programs [1] on a modified version of the Simplescalar simulator [2]. Our results show that the FSRAM improves the performance of a baseline processor with a 16KB data cache up to 55%, with an average of 9%. We also designed RTL and SPICE models of the FSRAM [3], which show that the FSRAM significantly improves memory access time, while reducing power consumption, with negligible area overhead.

1 Introduction

Rapid advances in high-performance computing architectures and semiconductor technologies have drawn considerable interest to high performance memories. Increases in hardware capabilities have led to performance bottlenecks due to the time required to access the memory. Furthermore, the on-chip memory performance in embedded systems directly affects designers' decisions about how to allocate expensive silicon area. Off-chip memory power consumption has become the energy consumption bottleneck as embedded applications become more data-centric.

Most of the recent research has tended to focus on improving performance and power consumption of on-chip memory structures [4, 5, 6] rather than off-chip memory. Moon *et al* [7] investigated a low-power sequential access on-chip memory designed to exploit the numerous sequential access patterns in digital signal processing (DSP) applications. Prefetching techniques from traditional computer architecture have also been used to enhance on-chip memory performance for embedded systems [8, 9, 10]. Other studies have investigated energy efficient off-chip memory for embedded systems, such as automatic data migration for multi-bank memory systems [11].

P.-C. Yew and J. Xue (Eds.): ACSAC 2004, LNCS 3189, pp. 88–101, 2004.

None of these previous studies, however, have investigated using off-chip memory structures to improve on-chip memory performance. This study demonstrates the performance potential of a novel, low-power, off-chip memory structure, which we call the *flexible sequential and random access memory* (FSRAM), to support flexible memory access patterns. In addition to normal random access, the FSRAM uses an extra "link" structure, which bypasses the row decoder, for sequential accesses. The link structure reduces power consumption and decreases memory access times; moreover, it aggressively prefetches data into the on-chip memory. In order to eliminate the potential data cache pollution caused by prefetching, a small fully associative *sequential access buffer* (SAB) is used in parallel with the data cache. VHDL and HSPICE models of the FSRAM have been developed to evaluate its effectiveness at the circuit level. Embedded multimedia applications are simulated to demonstrate its performance potential at the architecture level. Our results show significant performance improvement with little extra area used by the link structures.

The remainder of this paper is organized as follows. Section 2 introduces and explains the FSRAM and the SAB. In Section 3, the experimental setup is described. The architecture level performance analysis and area, timing and power consumption evaluations of the FSRAM are presented in Section 4. Section 5 discusses related work. Finally, Section 6 summarizes and concludes.

2 Flexible Sequential Access Memory

Our flexible sequential and random access memory (FSRAM) architecture is an extension of the sequential memory architecture developed by Moon, *et al* [7]. They argued that since many DSP applications have static and highly predictable memory traces, row address decoders can be eliminated. As a result, memory access would be sequential with data accesses determined at compile time. They showed considerable power savings at the circuit level.

While preserving the power reduction property, our work extends their work in two ways: (1) in addition to circuit-level simulations, we perform architectural-level simulations to assess the performance benefits at the application level; and (2) we extend the sequential access mechanism using a novel enhancement that increases sequential access flexibility.

2.1 Structure of the FSRAM

Fig. 1 shows the basic structure of our proposed FSRAM. There are two address decoders to allow simultaneous read and write accesses[1]. The read address decoder is shared by both the memory and the "link" structure. However, the same structure is used as the write decoder for the link structure, while the read/write decoder is required only for the memory. As can be seen, each memory word is associated with a link structure, an OR gate, a multiplexer, and a sequencer.

[1] Throughout the paper, all experiments are performed assuming dual-port memories. It is important to note that our FSAM does not *require* the memory to have two ports. The reason we chose two ports is that most modern memory architectures have multiple ports to improve memory latency.

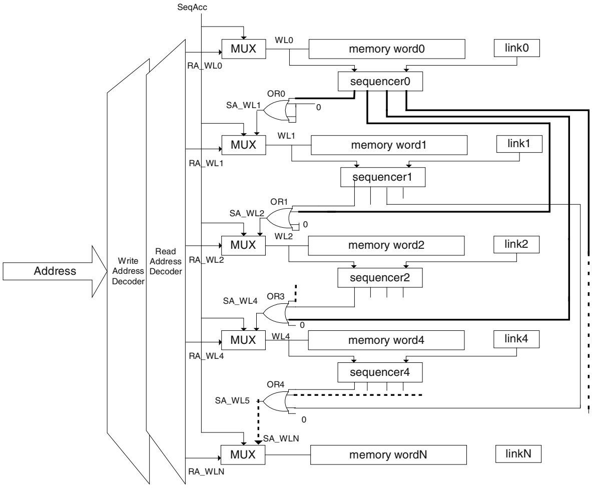


Fig. 1. The FSRAM adds a link, an OR gate, a multiplexer, and a sequencer to each memory word**.**

The link structure indicates which successor memory word to access when the memory is being used in the sequential access mode. With 2 bits, the link can point to four unique successor memory word lines (*e.g.*, N+1, N+2, N+4, and N+8). This link structure is similar to the "next" pointer in a linked-list data structure. Note that Moon *et al* [7] hardwired the sequencer cell of each row to the row below it. By allowing more flexibility, and the ability to dynamically modify the link destination, the row address decoder can be bypassed for many more memory accesses than previous mechanisms to provide greater potential speedup.

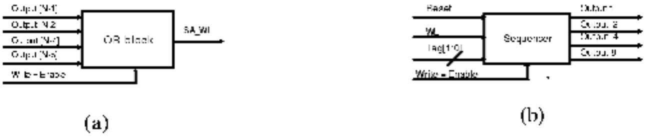


Fig. 2. (a) Block diagram of the OR block, (b) block diagram of the sequencer.

The OR block shown in Fig. 1 is used to generate the sequential address. If any of the four inputs to the OR block is high, the sequential access address (SA_WL) will be high (Fig. 2.a). Depending on the access mode signal (SeqAcc), the multiplexers choose between the row address decoder and the sequential cells. The role of the sequencer is to determine the next sequential address according to the value of the link (Fig. 2.b). If WL is high, then one of the four outputs is high. However if *reset* is high, then all four outputs go low irrespective of WL. The timing diagram of the signals in Fig. 2 is shown in our previous study [3].

The area overhead of the FSRAM consists of four parts - the link, OR gate, multiplexer, and sequencer. The overhead is in about the order of 3-7% of the total memory area for the word line size of 32 bytes and 64 bytes. More detailed area overhead results are shown in Table 2 in Section 4.2.

2.2 Update of the Link Structure

The link associated with each off-chip memory word line is dynamically updated using data cache miss trace information and run-time reconfiguration of the sequential access target. In this manner, the sequentially accessed data blocks are linked when compulsory misses occur. Since the read decoder for the memory is the same physical structure as the write decoder for the link structure, the link can be updated in parallel with a memory access. The default link value of the link is 0, which actually means the next line (2^0=1).

We note that the read and write operations to the memory data elements and the link RAM cells can be done independently. The word lines can be used to activate both the links and the data RAM cells for read or write (not all of the control signals are shown in Fig. 1).

There are a number of options for writing the link values:

1. The links can be computed at compile-time and loaded into the data memory while instructions are being loaded into the instruction memory.
2. The link of one row could be written while the data from another row is being read.
3. The link can be updated while the data of the same row is being read or written.

Option 1 is the least flexible approach since it exploits only static information. However, it could eliminate some control circuitry that supports the runtime updating of the links. Options 2 and 3 update the link structure at run-time and so that both exploit dynamic run-time information. Option 2, however, needs more run-time data access information compared to Option 3 and thus requires more control logic. We decided to examine Option 3 in this paper since the dynamic configuration of the links can help in subsequent prefetches.

2.3 Accessing the FSRAM and the SAB

In order to eliminate potential cache pollution caused by the prefetching effect of the FSRAM, we use a small fully associative cache structure, which we call the *Sequential Access Buffer* (SAB). In our experiments, the on-chip data cache and the SAB are accessed in parallel, as shown in Fig. 3. The data access operation is summarized in Fig. 4. When a memory reference misses in both the data cache and the SAB, the required block is fetched into the data cache from the off-chip memory. Furthermore, a data block pointed to by the link of the data word being currently read is pushed into the SAB if it is not already in the on-chip memory. That is, the link is followed and the result is stored in the SAB. When a memory reference misses in the data cache but hits in the SAB, the required block and the victim block in the data cache are swapped. Additionally, the data block linked to the required data block, but not already in on-chip memory, is pushed into the SAB.

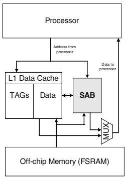


Fig. 3. The placement of the Sequential Access Buffer (SAB) in the memory hierarchy.

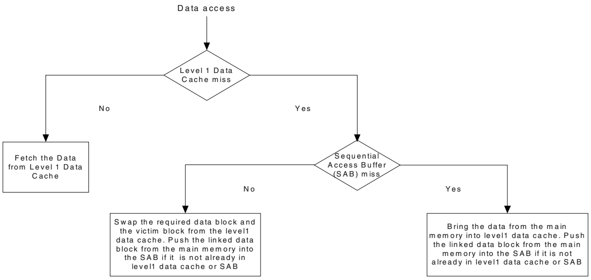

Data access
Level 1 Data Cache miss
No
Yes
Fetch the Data from Level 1 Data Cache
Sequential Access Buffer (SAB) miss
No
Yes
Swap the required data block and the victim block from the level1 data cache. Push the linked data block from the main memory into the SAB if it is not already in level1 data cache or SAB
Bring the data from the main memory into level1 data cache. Push the linked data block from the main memory into the SAB if it is not already in level1 data cache or SAB

Fig. 4. Flowchart of a data access when using the SAB and the FSRAM.

3 Experimental Methodology

To evaluate the system level performance of the FSRAM, we used SimpleScalar 3.0 [2] to run the Mediabench [1] benchmarks using this new memory structure. The basic processor configurations are based on Intel Xscale [12], The Intel XScale microarchitecture is a RISC core that can be combined with peripherals to provide applications specific standard products (ASSP) targeted at selected market segments. The basic processor configurations are as the following: 32 KB data and instruction L1 caches with 32-byte data lines, 2-way associativity and 1 cycle latency, no L2 cache, and 50 cycle main memory access latency. The default SAB size is 8 entries. The machine can issue two instructions per cycle. It has a 32-entry load/store queue and one integer unit, one floating point unit, and one multiplication/division unit, all with 1 cycle latency. The branch predictor is bimodal and has 128 entries. The instruction and data TLBs are fully associative and have 32 entries. The link structure in the off-chip memory was simulated by using a large enough table to hold both the miss addresses and their link values. The link values are updated by monitoring the L1 data cache miss trace. Whenever the gap between two continuous misses is 1x, 2x, 3x, 4x block line size, we update the link value correlated to the memory line that causes the first miss in the two continuous misses.

3.1 Benchmark Programs

We used the Mediabench [13] benchmarks ported to the SimpleScalar simulator for the architecture-level simulations of the FSRAM. We used four of the benchmark programs, *adpcm, epic, g721* and *mesa*, for these simulations since they were the only ones that worked with the Simplescalar PISA instruction set architecture.

Since the FSRAM link structure links successor memory word lines (Section 3.1), we show the counts of the address gap distances between two consecutive data cache misses in Table 1. We see from these results that the address gap distances of 32, 64, 128, 256 and 512 bytes are the most common, while the other address gap distances occur more randomly. Therefore, the FSRAM evaluated in this study supports address gap distances of 32, 64, 128 and 256 bytes for a 32-byte cache line, while distances of 64, 128, 256 and 512 bytes are supported for a 64-byte cache line.

For all of the benchmark programs tested, the dominant gap distances are between 32 and 128 bytes. Most of the tested benchmarks, except *g721*, have various gap distances distributed among 32 to 256 bytes. When the gap increases to 512 bytes, *epic* and *mesa* still exhibit similar access patterns while *adpcm* and *g721* have no repeating patterns at this gap distance.

Table 1. The frequencies (counts) of the various address distance gaps between two consecutive data cache misses for the tested benchmark programs

	adpcm encode	adpcm decode	epic encode	epic decode	g721 encode	g721 decode	mesa mipmap	mesa osdemo	Mesa texgen
32Bytes	121	121	167	82	609512	590181	78740	2212	229004
64 Bytes	7157	7157	3552	43	93	94	9	50896	22809
128 Bytes	979	979	1864	80	0	0	5	497	13441
256 Bytes	3237	3237	36	392	0	0	14	9	2
512 Bytes	0	0	5	896	0	0	3	1	16457

Another important issue for the evaluation of benchmark program performance is the overall memory footprint estimated from the cache miss rates. Table 2 shows the change in the L1 data cache miss rates for the baseline architecture as the size of the data cache is changed. In general, these benchmarks have small memory footprints, especially *adpcm* and *g721*. Therefore, we chose data cache sizes in these simulations to approximately match the performance that would be observed with larger caches in real systems. The default data cache configuration throughout this study is 16 KB with a 32-byte line and 2-way set associativity.

Table 2. The L1 data cache miss rates for the baseline architecture with various L1 cache sizes

	adpcm encode	adpam decode	epic encode	epic decode	g721 encode	g721 decode	mesa mipmap	mesa osdemo	Mesa Texge n
2KB	0.0214	0.0174	0.1424	0.1248	0.0010	0.0013	0.0894	0.0207	0.0735
4KB	0.001	0.0011	0.0703	0.0612	0.0003	0.0004	0.0444	0.0173	0.0337
8KB	0.0011	0.0011	0.0362	0.0591	0.0001	0.0001	0.0176	0.0142	0.0127
16KB	0.0010	0.0010	0.0162	0.0569	0.0000	0.0000	0.0086	0.0123	0.0068
32KB	0.0010	0.0010	0.0150	0.0535	0.0000	0.0000	0.0059	0.0112	0.0048

3.2 Processor Configurations

The following processor configurations are simulated to determine the performance impact of adding an FSRAM to the processor and the additional performance enhancement that can be attributed to the SAB.

orig: This is the baseline architecture with no link structure in the off-chip memory and no prefetching mechanism.

FSRAM: This configuration is described in detail in Section 3.1. To summarize, this configuration incorporates a link structure in the off-chip memory to exploit sequential data accesses.

FSRAM_SAB: This configuration uses the *FSRAM* with an additional small, fully associative SAB in parallel with the L1 data cache. The details of the SAB were given in Section 3.3.

tnlp: This configuration adds tagged next line prefetching [14] to the baseline architecture. With tagged next line prefetching, a prefetch operation is initiated on a miss and on the first hit to a previously prefetched block. Tagged next line prefetching has been shown to be more effective than prefetching only on a miss [15]. We use this configuration to compare against the prefetching ability of the *FSRAM*.

tnlp_PB: This configuration enhances the *tnlp* configuration with a small, fully associative Prefetch Buffer (PB) in parallel with the L1 data cache to eliminate the potential cache pollution caused by next line prefetching. We use this configuration to compare against the prefetching ability of the *FSRAM_SAB* configuration.

4 Performance Evaluation

In this section we evaluate the performance of an embedded processor with the FSRAM and the SAB by analyzing the sensitivity of the processor configuration *FSRAM_SAB* as the on-chip data cache parameters are varied. We also show the timing, area, and power consumption results based RTL and SPICE models of the FSRAM.

4.1 Architecture-Level Performance

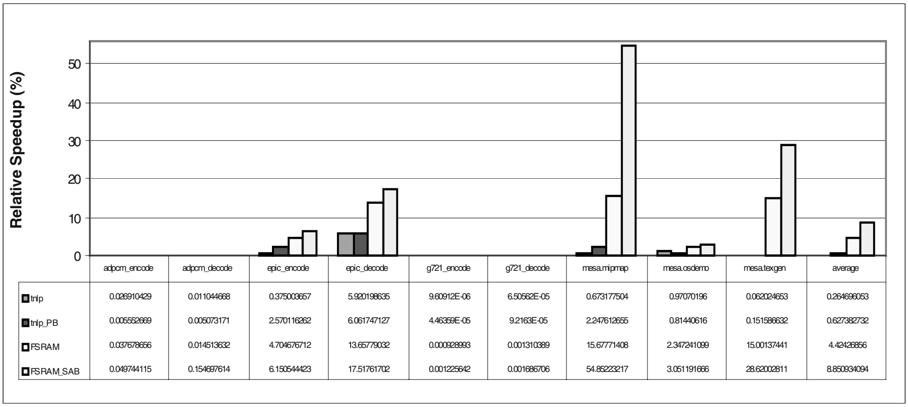
We first examine the *FSRAM_SAB* performance compared to the other processor configurations to show the data prefetching effect provided by the FSRAM and the cache pollution elimination effect provided by the SAB. Since the FSRAM improves the overall performance by improving the performance of the on-chip data cache, we evaluate the *FSRAM_SAB* performance while varying the values for different data cache parameters including the cache size, associativity, block size, and the SAB size.

Throughout Section 4.1, the baseline cache structure configuration is a 16 KB L1 on-chip data cache with a 32-byte data block size, 2-way associativity, and an 8-entry SAB. The average speedups are calculated using the execution time weighted average of all of the benchmarks [16].

Performance Improvement due to FSRAM. To show the performance obtained from the FSRAM and the SAB, we compare the relative speedup obtained by all four processor configurations described in Section 3.2 (i.e., *tnlp, tnlp_PB, FSRAM, FSRAM_SAB*) against the baseline processor configuration (*orig*). All of the processor configurations use a 16 KB L1 data cache with a 32-byte data block size and 2-way set associativity.

As shown in Fig. 5, the *FSRAM* configuration produces an average speedup of slightly more than 4% over the baseline configuration compared to a speedup of less than 1% for *tnlp*. Adding a small prefetch buffer (PB) to the *tnlp* configuration (*tnlp_PB)* improves the performance by about 1% compared to the *tnlp* configuration without the prefetch buffer. Adding the same size SAB to the FSRAM configuration (*FSRAM_SAB*) improves the performance compared to the FSRAM without the SAB by an additional 8.5%. These speedups are due to the extra small cache structures that eliminate the potential cache pollution caused by prefetching directly into the L1 cache. Furthermore, we see that the FSRAM without the SAB outperforms tagged next-line prefetching both with and without the prefetch buffer. The speedup of the FSRAM with the SAB compared to the baseline configuration is 8.5% on average and can be as high as 54% (*mesa_mipmap*).

Benchmark programs *adpcm* and *g721* have very small performance improvements, because their memory footprints are so small that there are very few data cache misses to eliminate in a 16KB data cache (Table 2) Never the less, from the statistics shown in Fig. 5, we can still see *adpcm* and *g721* follow the similar performance trend described above.



	adpcm_encode	adpcm_decode	epic_encode	epic_decode	g721_encode	g721_decode	mesa.mipmap	mesa.osdemo	mesa.texgen	average
▨ tnlp	0.026910429	0.011044668	0.375003657	5.920198635	9.60912E-06	6.50562E-05	0.673177504	0.97070196	0.062024653	0.264696053
■ tnlp_PB	0.005552669	0.005073171	2.570116262	6.061747127	4.46359E-05	9.2163E-05	2.247612655	0.81440616	0.151586632	0.627382732
□ FSRAM	0.037678656	0.014513632	4.704676712	13.65779032	0.000928993	0.001310389	15.67771408	2.347241099	15.00137441	4.42426856
□ FSRAM_SAB	0.049744115	0.154697614	6.150544423	17.51761702	0.001225642	0.001686706	54.85223217	3.051191666	28.62002811	8.850934094

Fig. 5. Relative speedups obtained by the different processor configurations. The baseline is the original processor configuration. All of the processor configurations use a 16KB data L1 cache with 32-byte block and 2-way associativity.

Parameter Sensitivity Analysis. We are interested in the performance of FSRAM with different on-chip data cache to exam how the off-chip FSRAM main memory structure improves on-chip memory performance. So in this section we study the effects of various data cache sizes (i.e., 2KB, 4KB, 8KB, 16KB, 32KB), data cache associativities (i.e., 1way, 2way, 4way, 8way), cache block sizes (i.e., 32 bytes, 64 bytes) and the SAB sizes (i.e., 4 entries, 8entries, 16entreis) on the performance. The baseline processor configuration through this section is the original processor configuration with a 2KB data L1 cache with 32-byte block size and 2-way associativity.

The Effect of Data Cache Size. Fig. 6 shows the relative speedup distribution among orig, tnlp_PB and FSRAM_SAB for various L1 data cache sizes (i.e., 2KB, 4KB, 8KB, 16KB, 32KB). The total relative speedup is FSRAM_SAB with a L1 data cache sizes over orig with a 2KB L1 data, which is divided into three parts: the relative speedup of orig with a L1 data cache size over orig with a 2KB L1 data cache; the relative speedup of tnlp_PB with a L1 data cache size over orig with a L1 data cache size; the relative speedup of FSRAM_SAB with a L1 data cache size over tnlp_PB with a L1 data cache size.

As shown, with the increase of L1 data cache size the relative speedup of *tnlp_PB* over *orig* decreases. FSRAM_SAB, in contrast, constantly keeps speedup on top of *tnlp_PB* across the different L1 data cache sizes. Furthermore, *FSRAM_SAB* even outperforms *tnlp_PB* with a larger size L1 data cache for most of the cases and on average. For instance, *FSRAM* with a 8KB L1 data cache outperforms *tnlp_PB* with a 32KB L1 data cache. However, *tnlp_PB* only outperforms the baseline processor with a bigger size data cache for *epic_decode* and *mesa_osdemo*.

The improvement in the performance can be attributed to several factors. While the baseline processor does not perform any prefetching, the tagged next line prefetching prefetches only the next word line. The fact that our method can prefetch with strides is one contributing factor in the smaller memory access time. Furthermore, prefetching is realized using sequential access, which is faster than random access. Another benefit is that prefetching with different strides does not require an extra large table to store the next address to be accessed.

tnlp_PB and *FSRAM_SAB* improve performance in the case that the performance of *orig* increases with the increase of L1 data cache size. However, they have little effect in the case that the performance of *orig* increases with the increase of L1 data cache size, which means the benchmark program has small memory foot prints (i.e., *adpcm, g721*). For adpcm, *tnlp* and *FSRAM_SAB* still improve performance when the L1 data cache size is 2K. For g721, the performance almost keeps the same all the time due to the small memory footprint.

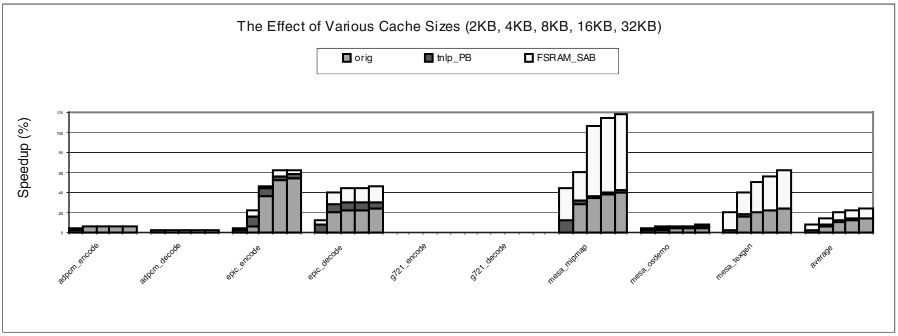


Fig. 6. Relative speedups distribution among the different processor configurations (i.e., *orig, tnlp_PB, FSRAM_SAB*) with various L1 data cache sizes (i.e., 2KB, 4KB, 8KB, 16KB, 32KB). The baseline is the original processor configuration with a 2KB data L1 cache with 32-byte block size and 2-way associativity.

The Effect of Data Cache Associativity. Fig. 7 shows the relative speedup distribution among orig, tnlp_PB and FSRAM_SAB for various L1 data cache associativity (i.e., 1way, 2way, 4way, 8way).

As known, increasing the L1 data cache associativity typically reduces the number of L1 data cache misses. The reduction in misses reduces the effect of prefetching from *tnlp_PB* and *FSRAM_SAB*. As can be seen, the performance speed up of *tnlp_PB* on top of *orig* decreases as the L1 data cache associativity increases. The speed up almost disappears when the associativity is increased to 8way for mesa_mipmap and mesa_texgen. However, FSRAM_SAB still provides significant speedups.

tnlp_PB and *FSRAM_SAB* still have little impact on the performance of adpcm and g721 because their small memory footprints.

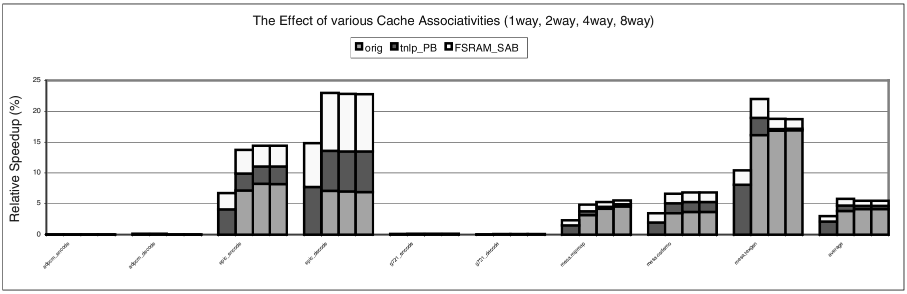


Fig. 7. Relative speedups distribution among the different processor configurations (i.e., *orig, tnlp_PB, FSRAM_SAB*) with various L1 data cache associativity (i.e., 1way, 2way, 4way, 8way). The baseline is the original processor configuration with a 2KB data L1 cache with 32-byte block size and 2-way associativity.

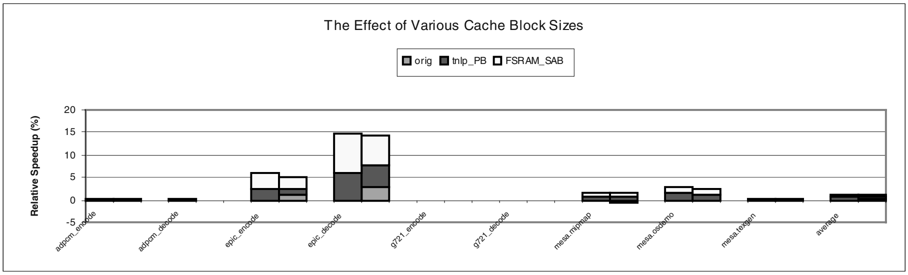


Fig. 8. Relative speedups distribution among the different processor configurations (i.e., *orig, tnlp_PB, FSRAM_SAB*) with various L1 data cache block sizes (i.e., 32B, 64B). The baseline is the original processor configuration with a 2KB data L1 cache with 32-byte block size and 2-way associativity.

The Effect of Data Cache Block Size. Fig. 8 shows the relative speedup distribution among *orig, tnlp_PB* and *FSRAM_SAB* for various L1 data cache block sizes (i.e., 32B, 64B).

As known increasing the L1 data cache block size typically reduces the number of L1 data cache misses. For all of the benchmarks the reduction in misses reduces the effect of prefetching from *tnlp_PB* and *FSRAM_SAB*. As can be seen, the performance speed up of *tnlp_PB* on top of *orig* decreases as the L1 data cache block size increases from 32-bytes to 64 bytes. However, the increasing of the L1 data cache block size can also cause potential pollutions as for *epic_encode* and *mesa_mipmap*. Tnlp with a small prefetching buffer reduces the pollution, and FSRAM_SAB further speeds up the performance.

The Effect of SAB Size. Fig. 9 shows the relative speedup distribution among orig, tnlp_PB and FSRAM_SAB for various SAB sizes (i.e., 4 entries, 8 entries, 16 entries).

Fig. 9 compares the FSRAM_SAB approach to a tagged next-line prefetching that uses the prefetch buffer that is the same size as SAB. As shown, FSRAM_SAB always add speedup on top of tnlp_PB. Further, *FSRAM_SAB* outperforms *tnlp* with a bigger size prefetch buffer. This result indicates that FSRAM_SAB is actually a more efficient prefetching mechanism than a traditional tagged next-line prefetching mechanism.

tnlp_PB and *FSRAM_SAB* still have little impact on the performance of adpcm and g721 because their small memory footprints.

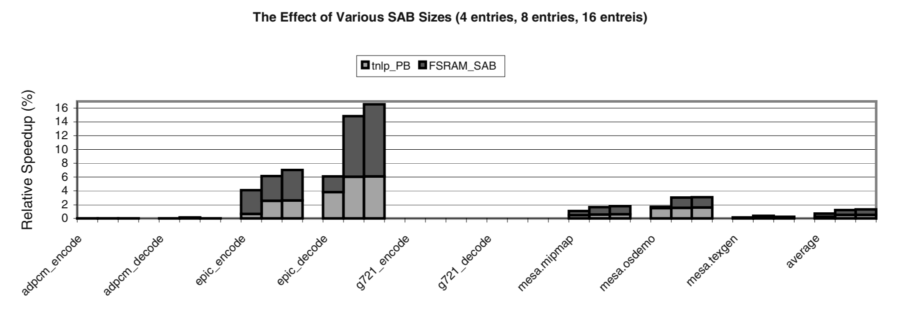


Fig. 9. Relative speedup distribution among the different processor configurations (i.e., *tnlp_PB*, *FSRAM_SAB*) with various SAB sizes (i.e., 4 entries, 8 entries, 16 entries). The baseline is the original processor configuration with a 2KB data L1 cache with 32-byte block size and 2-way associativity.

4.2 Timing, Area, and Power Consumption

We implemented the FSRAM architecture in VHDL to verify its functional correctness at the RTL level. We successfully tested various read/write combinations of row data *vs.* links. Depending on application requirements, one or two decoders can be provided so that the FSAM structure can be used as a dual-port or single-port memory structure. In all our experiments, we assumed dual-port memories since modern memory structures have multiple ports to decrease memory latency.

In addition to the RTL level design, we implemented a small 8x8 (8 rows, 8 bits per row) FSRAM in HSPICE using 0.18μm technology to test timing correctness and evaluate the delay of sequencer blocks. Note that unlike the decoder, the sequencer block's delay is independent of the size of the memory structure: it only depends on how many rows it links to (in our case: 4).

By adding sequencer cells, we will be adding to the area of the memory structure. However, in this section we show that the area overhead is not large, especially considering the fact that in today's RAMs, a large number of memory bits are arranged in a row. An estimate of the percentage increase in area was calculated using the formula $(\frac{A1}{A1-A2}-1)x100\%$ where A1 = Total Area and A2 = area occupied by the link, OR gate, MUX and the sequencer. Table 3 shows the results of the increases in area for different memory row sizes. The sequencer has two SRAM bits, which is not many compared to the number of bits packed in a row of the memory. We can see that the sequencer cell logic does not occupy a significant area either.

Table 3. Area overhead of FSRAM with various memory word line sizes

No. of bits per row of memory	Increase in area due to the MUX and the sequencer
8 (1 byte)	216%
16 (2 bytes)	119%
64 (8 bytes)	23.0%
256 (32 bytes)	7.12%
512 (64 bytes)	3.10%

As can be seen, the percentage increase in area drops substantially as the number of bits in each word line increases. Hence the area overhead is almost negligible for large memory blocks.

Using the HSPICE model, we compared the delay of the sequencer cell to the delay of a decoder. Furthermore, by scaling the capacity of the bit lines, we estimated the read/write delay and hence, calculated an overall speedup of 15% of sequential access compared to random access.

Furthermore, the power saving is 16% in sequential access at VDD = 3.3v in the 0.18 micron CMOS HSPICE model.

5 Related Work

The research related to this work can be classified into three categories: on-chip memory optimizations, off-chip memory optimizations, and hardware-supported prefetching techniques.

In their papers, Panda *et. al.* [4, 5] address data cache size and number of processor cycles as performance metrics for on-chip memory optimization. Shiue *et al.* [6] extend this work to include energy consumption and show that it is not enough to consider only memory size increase and miss rate reduction for performance optimization of on-chip memory because the power consumption actually increases. In order to reduce power consumption, Moon *et al.* [7] designed an on-chip sequential access only memory specifically for DSP applications that demonstrates the low-power potential of sequential access.

A few papers have addressed the issue of off-chip memory optimization, especially power optimization, in embedded systems. In a multi-bank memory system Dela Luz *et al.* [11] show promising power consumption reduction by using an automatic data migration strategy to co-locate the arrays with temporal affinity in a small set of memory banks. But their approach has major overhead due to extra time spent in data migration and extra power spent to copy data from bank to bank.

Zucker *et al.* [10] compared hardware prefeching techniques adopted from general-purpose applications to multimedia applications. They studied a stride prediction table associated with PC (program counter). A data-cache miss-address-based stride prefetching mechanism for multimedia applications is proposed by Dela Luz *et al.* [11]. Both studies show promising results at the cost of extra on-chip memory devoted to a table structure of non-negligible size. Although low-cost hybrid data prefetching slightly outperforms hardware prefetching, it limits the code that could benefit from prefetching [9]. Sbeyti *et. al.* [8] propose an adaptive prefetching mechanism which exploits both the miss stride and miss interval information of the memory access behavior of only MPEG4 in embedded systems.

Unlike previous approaches, we propose a novel off-chip memory with little area overhead (3-7% for 32 bytes and 64 bytes data block line) and significant performance improvements, compared to previous works that propose expensive on-chip memory structures. Our study investigated off-chip memory structure to improve on-chip memory performance, thus leaves flexibility for designer's to allocate expensive on-chip silicon area. Furthermore, we improved power consumption of off-chip memory.

6 Conclusions

In this study, we proposed the FSRAM mechanism that makes it possible to eliminate the use of address decoders during sequential accesses and also random accesses to a certain extent.

We find that FSRAM can efficiently prefetch the linked data block into on-chip data cache and improve performance by 4.42% on average for an embedded system using 16KB data cache. In order to eliminate the potential cache pollution caused by the prefetching, we used a small fully associative cache called SAB. The experiments show FSRAM can further improve the tested benchmark programs performances to 8.85% on average using the SAB. Compared to the tagged next-line prefetching, FSRAM_SAB constantly performs better and can still speedup performance when tnlp_PB cannot. This indicates that FSRM_SAB is more efficient prefetching mechanism.

FSRAM has both sequential accesses and random accesses. With the expense of negligible area overhead (3-7% for 32 bytes and 64 bytes data block line) from the link structure, we obtained a speedup of 15% of sequential access over random access from our designed RTL and SPICE models of FSRAM. Our design also shows that sequential access save 16% power consumption.

The link structure/configuration explored in this paper is not the only way; a multitude of other configurations can be used. Depending upon the requirement of an embedded application, a customized scheme can be adopted whose level of flexibility during accesses best suits the application. For this, prior knowledge of access patterns within the application is needed. In the future, it would be useful to explore power-speed trade-offs that may bring about a net optimization in the architecture.

References

[1] C.Lee, M. Potkonjak, and W. H. Mangione-Smith. "Mediabench: A tool for evaluating and synthesizing multimedia and communications systems." In Proc. of the 30th Annual International Symposium on Microarchitecture (Micro 30), December 1997

[2] Doug Burger and Todd M. Austin. "The simplescalar tool set version 2.0." Technical Report 1342, Computer Sciences Department, University of Wisconsin, June 1997.

[3] Ying Chen, Karthik Ranganathan, Amit Puthenveetil, Kia Bazargan, and David J. Lilja, “FSRAM: Flexible Sequential and Random Access Memory for Embedded Systems.” Laboratory for Advanced Research in Computing Technology and Compilers Technical Report No. ARCTiC 04-01, February, 2004.

[4] P. R. Panda, N. D. Dutt, and A. Nicolau. “Data cache sizing for embedded processor applications.” Technical Report TCS-TR-97-31, University of California, Irvine, June 1997.

[5] P. R. Panda, N. D. Dutt, and A. Nicolau. “Architectural exploration and optimizatioin of local memory in embedded systems.” International Symposium on System Synthesis (ISSS 97), Antwerp, Sept. 1997.

[6] W. Shiue, C. Chakrabati, “Memory Exploration for Low Power Embedded Systems.” IEEE/ACM Proc.of 36th. Design Automation Conference (DAC'99), June 1999.

[7] J. Moon, W. C. Athas, P. A. Beerel, J. T. Draper, “Low-Power Sequential Access Memory Design.”, IEEE 2002 Custom Integrated Circuits Conference, pp.741-744, Jun 2002.

[8] H. Sbeyti, S. Niar, L. Eeckhout, "Adaptive Prefetching for Multimedia Applications in Embedded Systems." DATE'04, EDA IEEE, 16-18 february 2004,Paris, France

[9] A. D. Pimentel, L. O. Hertzberger, P. Struik, P. Wolf, "Hardware versus Hybrid Data Prefetching in Multimedia Processors: A Case Study." in the Proc. of the IEEE Int. Performance, Computing and Communications Conference (IPCCC 2000), pp. 525-531, Phoenix, USA, Feb. 2000

[10] D. F. Zucker, M. J. Flynn, R. B. Lee, "A Comparison of Hardware Prefetching Techniques For Multimedia Benchfmarks." In Proceedings of the International Conferences on Multimedia Computing and Systems, Himshima, Japan, June 1996

[11] V. De La Luz, M. Kandemir, I. Kolcu, "Automatic Data Migration for Reducing Energy Consumption in Multi-Bank Memory Systems." DAC, pp 213-218, 2002

[12] Intel corparatin, "The intel XScale Microarchitecture technical summary", Technical report, 2001

[13] http://www.cse.psu.edu/~mdl/mediabench.tar.gz

[14] J. E. Smith, W. C. Hsu, “Prefetching in Supercomputer Instruction Caches.” In proceedings of Supercomputing92, pp. 588-597, 1992

[15] S. P. VanderWiel and D. J. Lilja, “Data Prefetch Mechanisms.” ACM Computing Surveys, Vol. 32, Issue 2, June 2000, pp. 174-199

[16] D. J. Lilja, “Measuring Computer Performance”, Cambridge University Press, 2000

Heuristic Algorithm for Reducing Mapping Sets of Hardware-Software Partitioning in Reconfigurable System

Seong-Yong Ahn[1], Jun-Yong Kim[2], and Jeong-A Lee[1]

[1] Chosun University, School of Computer Engineering, South Korea
{dis, jalee}@chosun.ac.kr
[2] Seoul National University, Dept. of Computer Engineering, South Korea
itecmdr@hanafos.com

Abstract. One of many technical challenges facing the designers of reconfigurable systems is how to integrate hardware and software resources. The problem of allocating each application function to general purpose processors (GPPs) and Field Programmable Gate Array (FPGAs) considering the system resource restriction and application requirements becomes harder. We propose a solution employing Y-chart design space exploration approach to this problem and develop Y-Sim, a simulation tool employing the solution. Its procedure is as follows: First, generate the mapping set by matching each function in a given application with GPPs and FPGAs in the target reconfigurable system. Secondly, estimate throughput of each mapping case in the mapping set by simulation. With the simulation results, the most efficient configuration achieving the highest throughput among the mapping cases would be chosen. We also propose HARMS (Heuristic Algorithm for Reducing Mapping Sets), a heuristic algorithm minimizing the mapping set by eliminating unnecessary mapping cases according to their workload and parallelism to reduce the simulation time overhead. We show the experimental results of proposed solution using Y-Sim and efficiency of HARMS. The experiment results indicates that HARMS can minimize the mapping set by 87.5% and most likely pick out the mapping case with the highest throughput.

1 Introduction

When one considers implementing a certain computational task, obtaining the highest performance can be achieved by constructing a specialized machine, i.e., hardware. Indeed, this way of implementation exists in the form of Application-Specific Integrated Circuits (ASICs). As, however, many reconfigurable devices such as Field Programmable Gateway Array(FPGA) are developed and improved, many computational tasks can be implemented, or configured, on those devices almost at any point by the end user. Their computation performance has not exceeded ASIC performance but could surpass general purpose processors by factor of several hundreds depending on applications. In addition to the performance gain, reconfigurable devices have the advantage of flexibility, contrary to the fact the structure of ASIC cannot be modified

P.-C. Yew and J. Xue (Eds.): ACSAC 2004, LNCS 3189, pp. 102–114, 2004.

after fabrication. Moreover, many independent computational units or circuits can be implemented on a single FPGA within its cell and connection limits and many FPGAs can be organized as a single system.

This novel technology brings about a primary distinction between programmable processors and configurable ones. The programmable paradigm involves a general-purpose processor, able to execute a limited set of operations, known as the instruction set. The programmer's task provids a description of the algorithm to be carried out, using only operations from this instruction set. This algorithm need not necessarily be written in the target machine language since compilation tools may be used; however, ultimately one must be in possession of an assembly language program, which can be directly executed on the processor in question. The prime advantage of programmability is the relatively short turnover time, as well as the low cost per application, resulting from the fact that one can reprogram the processor to carry out any other programmable task[1,3].

Reconfigurable Computing which promises performance and flexibility of systems has emerged as a significant area of research and development for both the academic and industrial communities. Still, there are many technical challenges facing the designers of reconfigurable systems. These systems are truly complex when we face how to integrate hardware and software resources[12]. Unfortunately, current design environments and computer aided design tools have not yet successfully integrated the technologies needed to design and analyze a complete reconfigurable system[11].

We propose a solution employing Y-chart design space exploration approach to this problem assuming pre-configuration policy and Y-Sim, a simulation tool for evaluating performance of the system. Its procedure is as follows: First, it generates the mapping set by matching each functions in given application with GPPs and FPGAs and estimates throughput of each mapping case in the mapping set by simulation. With the simulation results, the most efficient configuration achieving the highest throughput among the mapping cases would be chosen. During the simulation, Y-Sim resolves resource conflict problem arising when many subtasks try to occupy the same FPGA.

We also propose HARMS (Heuristic Algorithm for Reducing Mapping Sets), a heuristic algorithm reducing the huge size of a mapping set. Although each of mapping cases should be simulated to find an appropriate configuration, the elimination of unnecessary mapping cases according to the workload and parallelism helps to reduce the simulation time overhead significantly.

Section 2 summarizes previous related researches. Section 3 describes Y-Sim which employs Y-chart design space exploration approach. Section 4 explains HARMS and section 5 shows its efficiency with the minimization effect on simulation. Section 6 draws conclusions from the solution proposal and reduction algorithm.

2 Related Researches

Lee et al proposed reconfigurable system substructure targeting multi-application embedded system which executes various programs, such as PDA (Personal Digital

Assistant) and IMT2000 (International Mobile Telecommunication). They also proposed dynamic FPGA configuration model which interchanges several functional units on the same FPGA[5].

Keinhuis proposed Y-chart design space exploration[4]. This approach quantifies important performance metrics of system elements on which given applications will be executed. Keinhuis developed ORAS(Object Oriented Retargetable Architecture Simulator), a retargetable simulator, based on Y-chart approach[6]. ORAS calculates performance indicators of possible system configurations through simulation. However, simulation of reconfigurable system needs different simulation substructure other than ORAS since ORAS is limited to a specific calculation model of Stream-based Function.

Kalavade et al investigate the optimization problem in the design of multi-function embedded systems that run a pre-specified set of applications. They mapped given applications onto architecture template which contains one or more programmable processors, hardware accelerators and coprocessors. Their proposed model of hardware-software partitioning for multi-function system identifies sharable application functions and maps them onto same hardware resources. But, in this study, target applications is limited to a specific domain and FPGA is not considered[8].

Pruna et al. partitions huge applications into subtasks employing small size reconfigurable hardware and schedules data flows. Their study, however, is limited to one subtask - one FPGA configuration and multi-function FPGA is not considered[9].

3 Partitioning Tool

3.1 Design of Hardware Software Partitioning Tool

Y-Sim, a partitioning tool employing Y-chart approach, is designed to perform hardware-software partitioning for reconfigurable systems. This tool breaks down given application into subtasks and maps functions of each subtask and determines configuration of elements in the reconfigurable system. Performance indicators of each configuration is calculated to choose the best configuration. The same architecture model and application model in the Y-chart approach are employed and shown in Table 1.

The architecture model contains architecture elements(AEs) which indicate processing units such as general purpose processors and FPGAs. Each architecture element has its own AE_ID, and, in the case of FPGA, their maximum number of usable cells (Usable_Resource) and functional element (FE). Functional elements have resource request information which indicates the number of cells needed to configure the function. Functional elements also have execution time information which is needed for executing each data unit. In the case of FPGAs, those functions don't interfere others but do in the case of general purpose processors. Exclusion is considered in functional elements executed by general purpose processors. Fig. 1 shows an example of hardware model which has three architecture elements, each of which has two functional elements.

Table 1. Architecture Model and Application Model

Architecture Model	Application Model
Architecture := List_of_AE	Application := List_of_Pe
List_of_AE := AE | List_of_AE AE	List_of_PE := PE | List_of_PE PE
AE:=AE_ID Usable_Resources List_of_FE	PE:=PE_ID Next_PE_list FEQ_list
List_of_FE := FE | List_of_FE FE	Next_PE_list := PE_ID | Next_PE_list PE_ID
FE:=FE_ID Resource_Request Execution_Time Exclusion	FEQ_list := FEQ | FEQ_list FEQ

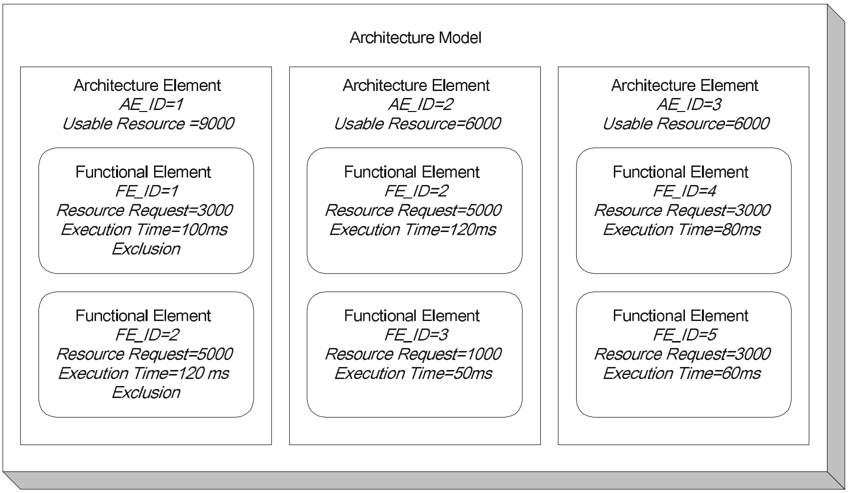


Fig. 1. An Example of Hardware Model

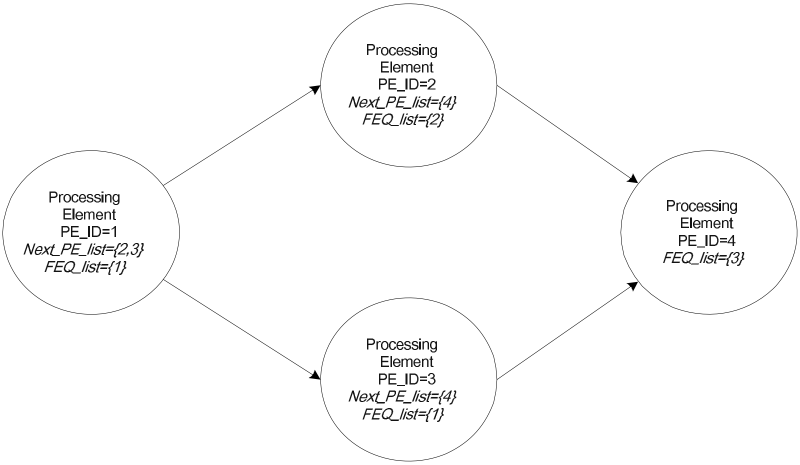


Fig. 2. An Example of Application Model

An application model is represented by processing elements(PE), subtasks of application and the direction of data flow indicated by Next_PE_list. The flow of data and the sequence of Processing Elements must have no feedback and no loop. Each process element has its own identifier, PE_ID and its function specified by FEQ_list. Fig. 2 shows an example of application model which has four process elements.

In Y-chart approach, mapping of Y-chart is done by generating mapping using architecture model and application model. A mapping set consists of mapping cases representing the configuration which directs Architecture elements to run identified functions of subtasks.

Y-Sim simulates each mapping cases in mapping set. Y-Sim handles resource conflict when many subtasks in given application simultaneously mapped on the same hardware element. The basic process of simulation is to let data units flow from the beginning of given application to the end of it. The measure of performance analysis, the timing information for each mapping is gathered when each Architecture Elements processes each data unit. Total elapsed time, one of the performance indicators, is calculated when all the given data units are processed. Other performance indicator includes throughput, the time taken to pass a data unit through the application, delay resulted from requested resource conflict, parallelism and usage of each architecture element. A system designer can choose the best mapping or the best system configuration from the performance indicators.

3.2 Structure of Y-Sim

Y-Sim consists of application simulator, architecture simulator and architecture-application mapping controller. The application simulator constructs application model from given application and simulates the flow of data units. The architecture simulator simulates each architecture element containing functional elements which executes requested function of data unit they received. The architecture-application mapping controller generates and maintains mapping information between subtasks and architecture elements and route of data unit from application subtasks to their mapped architecture elements. Fig. 3 shows the structure of Y-Sim constructed with an application shown before.

Application Simulator: The application simulator simulates the flow of data unit and controls the flow of data unit between processing elements which represent subtasks of given application. Each processing elements add their function request to data units received. The application simulator sends them to architecture-application mapping controller. Data units routed to and processed in architecture simulator are returned to the processing element and the processing element sends the data units to the next process. The application simulator generates data units and give them to the first processing element of generated application model and gathers processed data units to calculate the time taken to process a data unit with given application model and delay.

Architecture Simulator: The architecture simulator simulates the service of functional element requests from application simulator and calculates execution time taken to execute each data unit. When receiving a data unit, a processing element in application simulator invokes application-architecture mapping controller to ask that a functional element in an architecture element processes the data unit and other request should wait until the request is served. In addition, when the exclusion flag in some functional elements is set, the architecture element exclusively executes the functional elements. Because of the exclusion, hardware resource request conflicts are occurred under the many subtask-to-one architecture element configuration and many subtask-to-one functional element in an architecture element configurations. In fig 3, for example, resource conflict would occur in some mapping because of subtask 1 and 3 issuing the functional element request to the same functional element. Y-Sim takes into account the conflict and resolves it. After serving the request, the functional elements record execution time in the data unit and return it to the application simulator, which returns it to the processing element the data unit originated from. Each architecture element records its time for each mapping case.

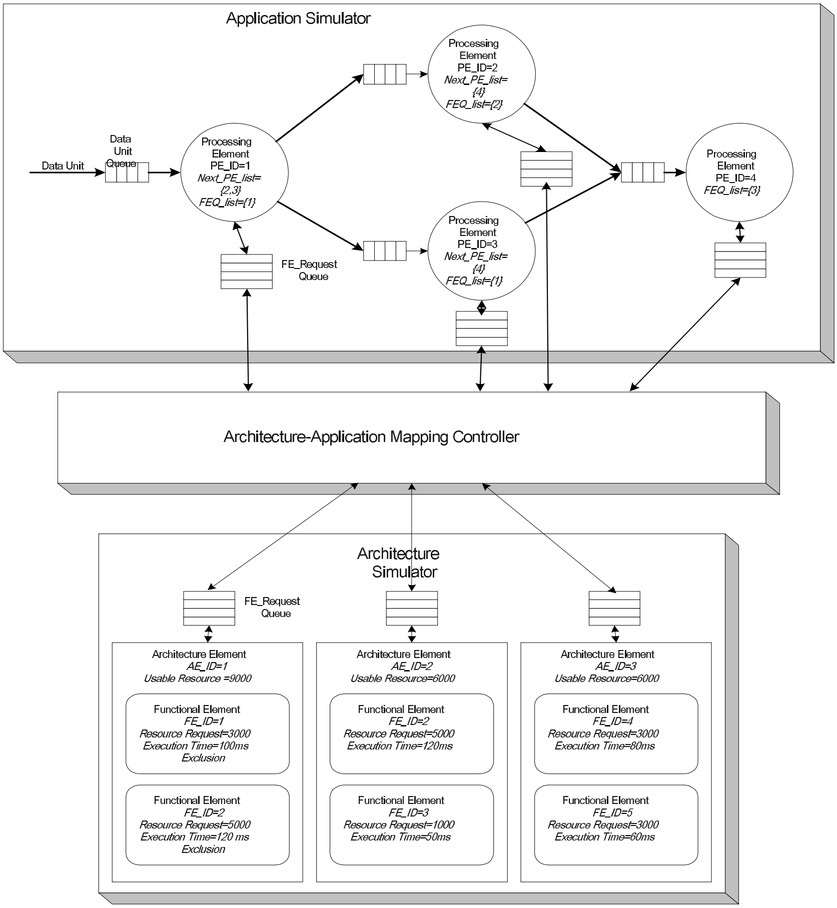

Application Simulator
Data Unit
Data Unit Queue
Processing Element PE_ID=1 Next_PE_list={2,3} FEQ_list={1}
FE_Request Queue
Processing Element PE_ID=2 Next_PE_list={4} FEQ_list={2}
Processing Element PE_ID=3 Next_PE_list={4} FEQ_list={1}
Processing Element PE_ID=4 FEQ_list={3}
Architecture-Application Mapping Controller
Architecture Simulator
FE_Request Queue
Architecture Element AE_ID=1 Usable Resource =9000
Functional Element FE_ID=1 Resource Request=3000 Execution Time=100ms Exclusion
Functional Element FE_ID=2 Resource Request=5000 Execution Time=120 ms Exclusion
Architecture Element AE_ID=2 Usable Resource=6000
Functional Element FE_ID=2 Resource Request=5000 Execution Time=120ms
Functional Element FE_ID=3 Resource Request=1000 Execution Time=50ms
Architecture Element AE_ID=3 Usable Resource=6000
Functional Element FE_ID=4 Resource Request=3000 Execution Time=80ms
Functional Element FE_ID=5 Resource Request=3000 Execution Time=60ms

Fig. 3. The Structure of Y-Sim

Architecture-Application Mapping Controller: The architecture-application mapping controller generates all the possible mapping using information of subtask and functional elements of each architecture element and subtask information of given application. The size of mapping set is Πij where i is the number of functions which subtasks of given application executes and j is the number of possible functional elements which can execute functions of subtasks. The size of mapping set in Fig. 3 is 2 since function 2 requested by subtask 2 can be executed in architecture element 1 and architecture element 2. Y-Sim iterates each mapping case and during the simulation, the architecture-application mapping controller routes data units sent by processing elements in the application simulator to corresponding architecture elements in architectural simulator. Performance indicators of each iteration are gathered and used to choose the best mapping.

4 Simulation Time Reduction Heuristic

The architecture-application mapping controller generates Πij mapping. Therefore, as the number of subtasks of given application and the number of functional request of each subtask increases and as more architecture elements and functional elements in them are accommodated, the size of mapping set increases exponentially. The performance of simulator is directly influenced by the size of mapping set and can be improved if the size of mapping set is reduced before simulation. The simulation time reduction algorithm basically decreases the size of mapping set. The simulation time reduction algorithm, HARMS, decrease the size as follows: The algorithm first excludes impossible mapping cases by computing the available resources of FPGA (the number of FPGA cells) and requested number of FPGA cells in functional elements. Then it reduces mapping set by analyzing workload-time taken to process a data unit, parallelism and their relationship which can be identified when each mapping case is generated.

4.1 Mapping Set Reduction Based on Resource Restriction

Various functions can be configured and executed simultaneously on the same FPGA. However, because FPGAs have restricted number of programmable cells, these functions should be organized or synthesized to fit in the size of target FPGA. For example, XC5202 from Xilinx has 256 logic cells and the circuit size of below 3,000 gates can be configured. XC5210 has 1296 logic cells and logic circuit from the size of 10000 gates to 16000 gates can be configured. [10]

Because of this resource restriction, some mapping cases generated in architecture-application mapping controller are impossible to be configured. The reduction is done by comparing the number of cells in each FPGA of target system and the number of FPGAs cell required to configure each mapping case.

4.2 Mapping Set Reduction with Analysis of Workload Parallelism

Reconfigurable Systems improve system performance by configuring co-processors or hardware accelerators on reconfigurable devices such as FPGA. When a FPGA and a general purpose processor execute the same function, FPGA has better performance even though their shortcoming of low clock speed.[1,3]. In addition, reconfigurable systems have higher parallelism because functions executed on a FPGA also can be executed on a general purpose processors and simultaneously configured functions on the same FPGA without interfering each other. In a reconfigurable system using FPGA, Parallelism, *P1* and *P2*, Workload, *W1* and *W2* and throughput *T1* and *T2* of *m1* and *m2*, two mapping cases in a mapping set, *M* has following relationship.

$$\textit{IF } W1>W2 \textit{ and } P1<P2 \textit{ then } T1<T2 \tag{1}$$

Throughput of a configuration accelerated with increased parallelism and decreased workload by FPGA is higher than a configuration without them. Heuristic Algorithm for Reducing Mapping Sets (HARMS) is based on this relationship. As shown later, HARMS exclude mapping cases which have more workload and less parallelism than any other mapping case. HARMS is a heuristic algorithm because some specific mapping cases have workload and parallelism which do not satisfy the equation 1 and HARMS exclude such mapping cases.

5 Experimental Results

Section 5 shows the effectiveness and efficiency of the proposed algorithm, HARMS. With simulation results, the efficiency of HARMS is presented showing the percentage of inefficient mapping cases that HARMS can eliminate. A specific case that HARMS excludes the best mapping cases is also represented and its reason is explained.

5.1 Simulation Environments

We assume that the architecture model is based on a reconfigurable system which has a general purpose processor and several FPGA and the application model is based on H.263 decoder algorithm shown in Fig. 4 and picture in picture algorithm shown in Fig. 5 used to display two images simultaneously on a television screen. A data flow of the H.263 decoder is modeled to encompass a feedback loop so as the application to be modeled as a streaming data processing model. The H.263 is a widely used for video signal processing algorithm and its data flow is easy to identify[13]. Some subtasks in picture in picture algorithm request the same functions and resource conflict usually arise when implemented on reconfigurable system. Therefore, these two algorithms are appropriate to evaluate the performance of Y-Sim and the efficiency of

HARMS. For each algorithm the reconfigurable system of architecture model was simulated and mapping cases was generated by Y-Sim.

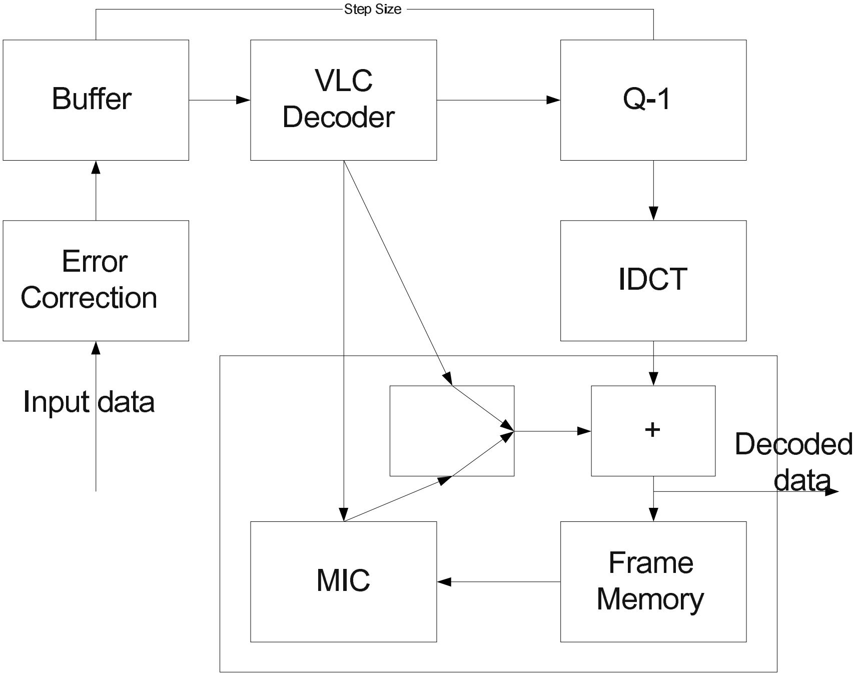


Fig. 4. Picture in Picture Algorithm

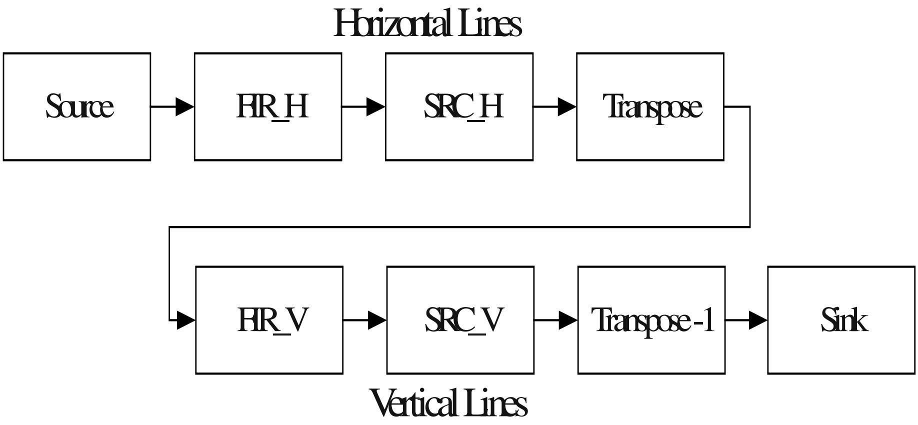


Fig. 5. Picture in Picture Algorithm

5.2 Efficiency and Accuracy of HARMS

In order to show the efficiency and accuracy of HARMS, simulation result illustrates that excluded mapping cases are not mostly the best mapping cases. The accuracy of HARMS is shown as follows: First, we assume that the generated mapping cases of H.263 decoder and picture in picture run on a reconfigurable system with one general purpose processor and three FPGA. Then, the accuracy of HARMS is proved by showing that reduced mapping set has the best mapping cases. Since HARMS doesn't analyze the structure of mapping cases, the best mapping case is excluded under some specific system configuration. For these cases, a solution resolving this problem is also presented.

5.2.1 Comparison of Before Running HARMS and After

Fig. 6 and Fig. 7 illustrate comparison of mapping set before and after applying HARMS. The initial size of the mapping set of picture in picture model is 16384 and the size is reduced to 1648 by resource restriction. HARMS reduce the size of the mapping set to 240 and the reduced mapping set contains the best mapping case. As the simulation results shows, HARMS reduces mapping set without excluding the best mapping case.

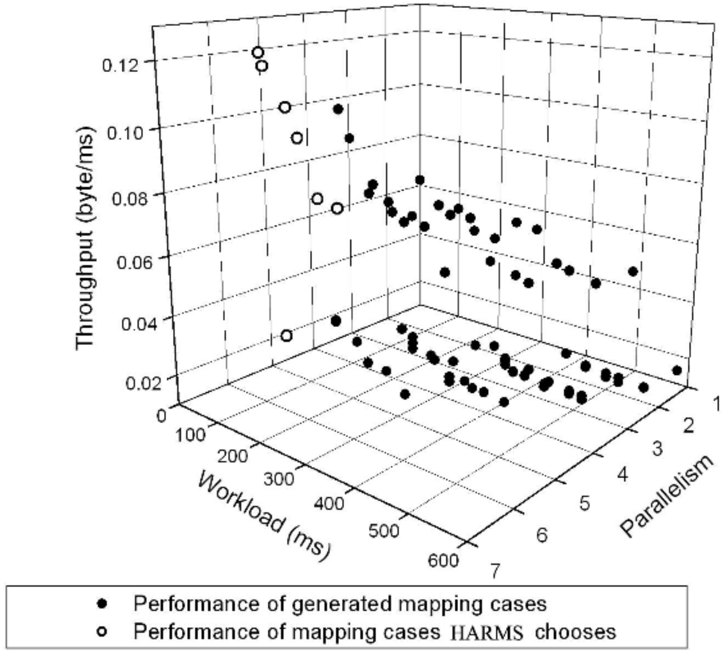


Fig. 6. Comparison of mapping set of H.263 before and after running HARMS

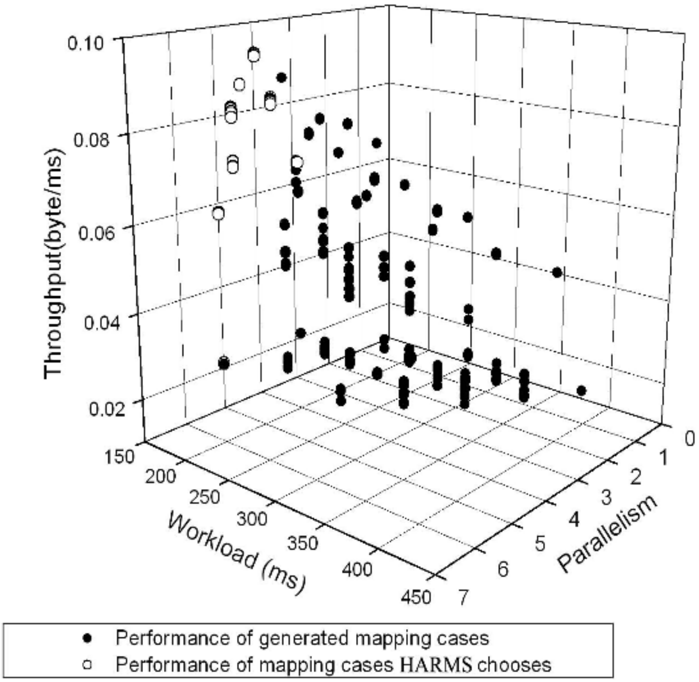


Fig. 7. Comparison of mapping set of Picture in Picture before and after running HARMS

Table 2. The Efficiency of HARMS

	Initial size of mapping set	Resource limitation is considered	HARMS is applied	
			size	efficiency(%)
H.263(1)	4096	4096	928	77.3
H.263(2)	729	624	186	70.2
H.263(3)	64	40	7	82.5
H.263(4)	64	64	8	87.5
H.263(5)	4096	4084	2175	46.7
Pip(1)	64	13	12	7.7
Pip(2)	16384	1648	240	85.4

5.3 Efficiency of HARMS

This section shows the reduction ratio of mapping set when HARMS is applied. Table 2 shows simulation and reduction results of five system configurations of H.263 decoder and two cases of picture in picture algorithm. Each system configuration is as follows:

H.263 (1) - two general purpose processors (GPPs) and two FPGAs.
H.263 (2) - one GPP and one FPGA.
H.263 (3) - one GPP and one FPGA.
H.263 (4) - Similar to H.263 (3) but smaller FPGA.
H.263 (5) - one GPP and three FPGAs.
Pip(1)-one GPP and one FPGAs,
Pip(2)-one GPP and three FPGAs,

The efficiency is the ratio of the size of mapping set HARMS applied on (Aft_size) to the size of mapping set resource limitation is considered(Bef_size). The following shows its equation

$$\textbf{\textit{Efficiency of HARMS}} = (\textit{Bef\_size} - \textit{Aft\_size})/\textit{Befsize} \qquad (2)$$

As shown in table 2, HARMS reduces the size of mapping set by about 80 percent. In the case of H.263(5), architecturally same mapping cases appeared on different mapping cases in the generated mapping set since the system has three same FPGAs of the same size on which the same functional element can be configured. As a result of redundant configuration, the efficiency of HARMS is comparatively low, 46.7%. In the case of Pip(1), the size of mapping set is minimized by resource restriction that the efficiency is very low, 7.7%.

6 Conclusion

This paper proposes Y-Sim, a hardware-software partitioning tool. With given application model and architecture model, representing configuration of reconfigurable system, Y-Sim explores possible mapping between architecture model and application model and generate mapping set which represents possible mapping set of given reconfigurable system. Then, Y-Sim simulates each mapping case in the mapping set and finds the best one. Y-Sim takes resource restriction of FPGA such as the number of usable cells into account and resolves resource conflicts arising when many subtasks of given application share the same resource.

As more FPGAs are installed in a reconfigurable system and more functions are configured on FPGAs, subtasks can be mapped on many FPGAs or general purpose processors. Consequently, the size of mapping set increases exponentially and the time taken to analyze performance of such system is heavily influenced by the size. Generally, throughput of reconfigurable system using FPGA has positive relationship to parallelism and negative relationship to workload. Based on this relationship, proposed heuristic algorithm, HARMS efficiently reduces mapping set. Experimental result shows that HARMS can reduce the size of the mapping set. For the various architecture and application model of reconfigurable system running H.263 decoder and picture in picture algorithm, HARMS reduces the size of mapping set by about 80% and up to 87.5%.

Acknowledgement. This work was supported by grant No. R04-2002-000-00081-0 from the research program of the KOSEF.

References

1. O.T. Albahama, P. Cheung, and T.J. Clarke, "On the Viability of FPGA-Based Integrated Coprocessors," In Proceedings of IEEE Symposium of FPGAs for Custom Computing Machines, pp. 206-215, Apr. 1996
2. J-L, Gaudiots. "Guest Editors' Introduction", IEEE Transactions on Computers, VOL.48, No.6, June 1999.
3. E. Sanchez. M. Sipper, J.-O. Haenni, J.-L. Beuchat, A. Stauffer, and A. Perez-Uribe, "Static and Dynamic Configurable Systems", IEEE Transactions on Computers VOL.48, No.6, June 1999.
4. B. Kienhuis, E. Deprettere, K.A. Vissers, and P. Wolf. An approach for quantitative analysis of application-specific dataflow architectures. In Proceedings of 11th Intl. Conference of Applications-specific Systems, Architectures and Processors (ASAP'97), pages 338-349, Zurich, Switzerland, 1997
5. S. Lee ; K.Yun ; Choi, Kiyoung ; S. Hong ; S. Moon ; J. Lee, "Java-based programmable networked embedded system architecture with multiple application support.", *International Conference on Chip Design Automation*, pp.448-451, Aug. 2000.
6. A.C.J. Kienhuis, Design Space Exploration of Stream-based Dataflow Architectures, PhD thesis, Delft University of Technology, Netherlands, 1998.
7. S. Bakshi, D. D. Gajaski, "Hardware/Software Partitioning and Pipelining", In Proceedings of the 34th annual conference on Design Automation Conference, 1997, pages 713-716
8. A. Kalavade, P. A. Subrahmanyam, "Hardware/Software Partitioning for Multifunction Systems", In Proceedings of International Conference on Computer Aided Design, pages 516-521, 1997,
9. K.M. Gajjala Purna and D. Bhatia "Temporal Partitioning and Scheduling Data Flow Graphs for Reconfigurable Computers" IEEE Transactions on Computers, VOL.48, No.6, June 1999.
10. XC5200 Series Field Programmable Gate Arrays Databook, ver 5.2, Xilinx Inc, Nov 1998.
11. S. Mohanty, V.K. Prasanna, S. Neema, J. Davis, “Rapid Design Space Exploration of Heterogeneous Embedded Systems using Symbolic Search and Multi-Granular Simulation”, LCTES’02-SCOPES’02, Berlin, Germany, June 2002.
12. Katherine Compton, Scott Hauck, “Reconfigurable Computing: A survey of Systems and Software” ACM Computing Surveys, Vol.34, No.2 June 2002, pp.171-210
13. http://www.itut.int/itudoc/rec/h/h263.html

Architecture Design of a High-Performance 32-Bit Fixed-Point DSP

Jian Chen, Ruhao Xu, and Yuzhuo Fu

School of Microelectronics, Shanghai Jiao Tong University
1954 Huashan Road, Shanghai 200030, China
{chenjian, xuruhao, fuyuzhuo}@ic.sjtu.edu.cn

Abstract. In this paper, the architecture of a high-performance 32-bit fixed-point DSP called DSP3000 is proposed and implemented. The DSP3000 employs super-Harvard architecture and can issue three memory access operations in a single clock cycle. The processor has eight pipe stages with separated memory read and write stages, which alleviate the data dependency problems and improve the execution efficiency. The processor also possesses a modulo addressing unit with optimized structure to enhance the address generation speed. A fully pipelined MAC (Multiply Accumulate) unit is incorporated in the design, which enables $32 \times 32 + 72$ MAC operation in a single clock cycle. The processor is implemented with SMIC $0.18\mu m$ 1.8V 1P6M process and has a core size of 2.2mm by 2.4mm. Test result shows that it can operate at a maximum frequency of 300MHz with the average power consumption of $30mw/100MHz$.

1 Introduction

Digital signal processor finds its applications in a wide variety of areas, such as wireless communication, speech recognition and image processing, where high speed calculation capability is of primary concern. With the ever increasing applications in battery-powered electronics, such as cellular phones and digital cameras, the power dissipation of DSP is also becoming a critical issue. The trend of achieving high performance mean while preventing power consumption from surging is imposing a great challenge on modern DSP design.

Various approaches have been proposed to address the challenge by exploring different levels and aspects in the entire DSP design flow.[1][2] This paper, however, would focus its efforts on the architecture level design of a DSP to achieve both performance and power consumption requirements. It presents the overall architecture of DSP3000, a high-performance 32-bit fixed-point digital signal processor. It also proposes novel micro-architectures aiming at increasing the performance and reducing chip area. Techniques to reduce power consumption of the chip are described as well.

The rest of the paper is organized as follows. Section 2 gives a detailed description of the overall architecture of the processor as well as the micro-architectures

P.-C. Yew and J. Xue (Eds.): ACSAC 2004, LNCS 3189, pp. 115–125, 2004.

in pipeline organization, address generation unit and MAC unit. Section 3 describes the performance of the implemented processor. Section 4 presents the conclusion of the research.

2 Architecture of DSP3000

In order to enhance the performance, the design of the processor incorporates the ideas of deep pipelining, two-way superscalar and super-Harvard structure. Specifically speaking, the core of the DSP has eight pipe stages and two datapaths working in parallel. It is also capable of accessing data in two different data memory spaces simultaneously. Fig. 1 illustrates the major components of the processor, which is composed of the DSP core, on-chip memories, instruction cache and peripherals.

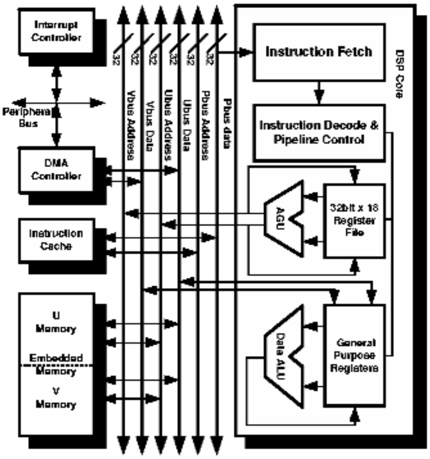


Fig. 1. The block diagram of DSP3000

The instructions are fetched from instruction cache by the instruction fetch unit and passed to the instruction decode unit,where the instructions are decoded into microcodes. This unit also has a hardware stack with a depth of sixteen to effectively support hardware loops. The decoded instructions are issued to AGU (Address Generation Unit) and DALU (Data Arithmetic Logic Unit) for corresponding operations. AGU is responsible for the calculation of the addresses of both U and V memories that the core is going to access. Various addressing modes, including register direct addressing mode and modulo addressing mode, are supported by this module. Among them, modulo addressing mode is particularly important to the performance of the DSP and will be examined in detail in section 2.2. DALU works in parallel with AGU and accomplishes the arithmetic and logic operations such as MAC (Multiply Accumulate), accumulation and shift. Besides, it also offers some powerful bit operations, such as bit set and bit clear, to enhance the control capability of the chip so that the chip can also be applied as an MCU (Micro Control Unit). The operation of MAC, critical to the performance of DSP algorithms, is implemented with a novel structure, which makes the chip finish MAC operation in one clock cycle and reduces the chip area and the power consumption at the same time. This structure will be highlighted in section 2.3.

2.1 Pipeline Organization and Write Back Strategy

The data-path of DSP3000 core is divided into eight pipe stages with their names P1-P8 respectively, as is shown in Fig. 2. In order to coordinate the operations between AGU and DALU, the execution actually occupies four pipe stages, i.e., P4-P7. Basically, P4 and P5 in DALU simply pipe the instructions down and do no extra operations except for some decode activities to prepare for the operations begin on P6.The pipeline organization features the separated memory read and write stages in AGU, which are introduced to deal with the data dependence problems aggravated by the wide span of execution pipe stages. This could be explained with the following instructions:

```
MAC R0, R1, R5 U: (AR0+), R0 V: (AR4+), R1
SUB R2, R4 U:(AR0+), R2 R5, V: (AR4+)
```

The first instruction accomplishes MAC operation with the data in register R0, R1 and R5 and stores the result into register R5. Meanwhile two parallel move operations load R0 and R1 with the data read from U and V memory respectively. The second instruction subtracts R4 from R2 and stores the result into R4. Also there are two parallel move operations in this instruction. The second one of them stores the value of register R5, which is the result of the first instruction, to V memory. If memory read and write are combined in a single stage, this operation must start on the beginning of P6 stage since arithmetic operation begins on this stage. Otherwise DALU may not get the desired data from memory for corresponding operations. Yet, the desired value of R5 is not available until the first instruction finishes the operation on P7. This would introduce a stall or a NOP between the two instructions to avoid the data hazard, reducing the instruction execution efficiency. With the separated memory read and write strategy, however, the memory write operation happens on P7 stage. The above data hazard can then be avoided by forwarding the accumulate result to the input of memory write function unit.

The new memory access strategy, however, would give rise to memory access contention problems if no other measures are taken. This problem can be illustrated by the following instructions:

```
MOVE R3,U:(AR6)
MOVE U:(AR7),R4
```

The first instruction writes the U memory indexed by AR6 with the value in R3, while the second instruction reads the data from U memory to R4. These two

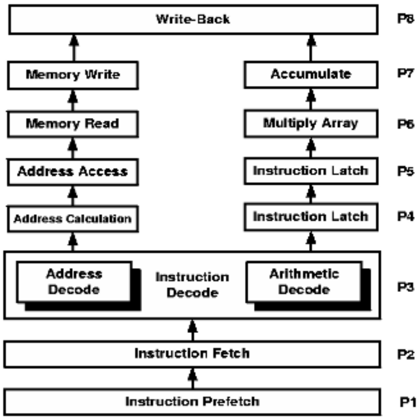


Fig. 2. The pipeline structure of DSP3000 core

instructions will cause the DSP core to access the U memory in different pipe stages simultaneously. In order to avoid this contention, the pipeline incorporates a buffer called write back queue in the write back stage P8, as is illustrated in Fig. 3. When the control unit detects memory access contentions, it holds the memory write operation and stores the corresponding data into the queue while continues the memory read operation. That is to say, the memory read instruction has a higher priority than the preceding memory write instruction. The memory write operation can be resumed as soon as the address bus is free.In order to prevent RAW(Read after Write) hazard during the execution of memory read operation, AGU first searches the write back queue for the desired data. If there is a hit, the data can be fetched directly from the write back queue. Otherwise, it goes to memory to continue the read operation.

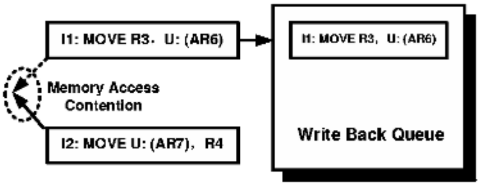


Fig. 3. The mechanism of write back queue

The depth of the write back queue is one, which could be proved by the following analysis. The contention problem could only happen in a read after write situation. If the following instruction is a read operation, the write operation can be still suspended in the write back queue and no new write instruction is pushed into this queue. On the other hand, if the following instruction is a write operation, then the memory bus would be free for one clock cycle, during which the suspended write operation can be finished. Consequently, the write back queue is only one level deep, which will not lead to the hardware overhead. Besides the benefit of increasing instruction execution efficiency, the proposed pipeline organization also contributes to the saving of the power that would otherwise be consumed on NOP instructions.

2.2 Modulo Addressing Unit in AGU

Modulo addressing is widely used in generating waveforms and creating circular buffers for delay lines.[6] The processor supports two kinds of modulo addressing, i.e., increment modulo addressing and decrement modulo addressing. Conventionally, they are realized by creating a circular buffer defined by the lower boundary (base address) and the upper boundary (base address + modulus - 1).When the calculated address surpasses the upper boundary, it wraps around through the lower boundary. On the other hand, if the address goes below the lower boundary, it wraps around through the upper boundary.[6] The modulo addressing unit of the processor adopts this basic idea, yet it employs a modified approach to improve the efficiency.

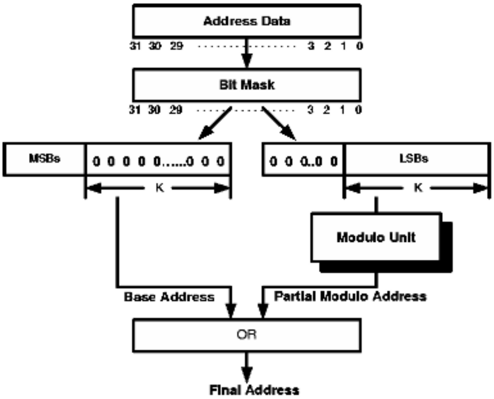


Fig. 4. The modified modulo addressing scheme

As is shown in Fig. 4, a bit mask generated according to value of modulus divides the pre-calculated address data into two sections, the MSBs(Most Significant Bit) and the LSBs(Least Significant Bit). The MSBs, together with the succeeding K-bit zeros, form the base address of the circular buffer, where $2K > modulus$ and $2K - 1 <= modulus$. The K-bit LSBs are zero extended to 32 bits before they are put into the Modulo Unit, where the modulo operation is performed. The base address and the partial modulo address generated in modulo unit are then ORed together to get the final address. With this scheme, the low boundary of the circular buffer becomes zero from the Modulo Unit's

perspective. Consequently, the following algorithm can be applied in the Modulo Unit.
For increment modulo addressing:

```
Intermediate = LSBs + OffsetValue;
If (Intermediate > Modulus)
PartialModuloAddress=Intermediate-Modulus;
Else
PartialModuloAddress=Intermediate;
```

For decrement modulo addressing:

```
Intermediate = LSBs - OffsetValue;
If (Intermediate < 0)
PartialModuloAddress=Intermediate+Modulus;
Else
PartialModuloAddress=Intermediate;
```

It is required that OffsetValue should be no larger than Modulus, or the result is unpredictable.[6] The corresponding hardware implementation of increment modulo addressing is shown in Fig. 5(a). The longest path of this structure includes an adder, a subtracter and a multiplexer. Yet, further optimization on the unit can be carried out by dividing the timing critical path into two parallel data-paths to calculate the two possible addresses simultaneously, as is illustrated below:

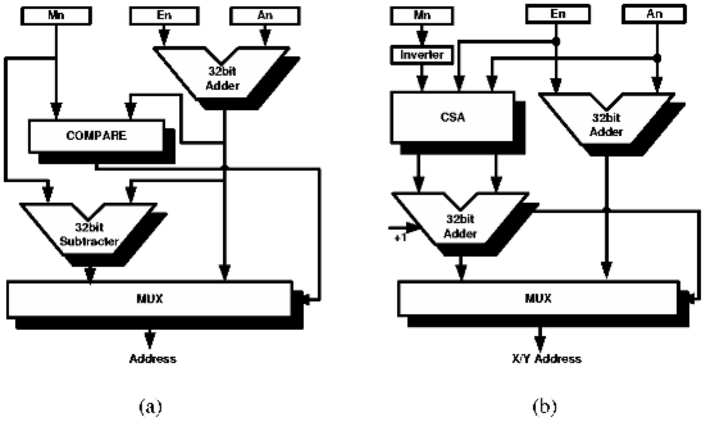


Fig. 5. The diagram of the modulo unit. An, En and Mn are the registers holding the LSBs , the offset value and the modulus respectively. (a).The diagram of the direct implementation of the unit. (b).The diagram of the proposed implementation of the unit, where CSA stands for Carry Save Adder.

Datapath 1: $LSBs + OffetValue$
Datapath 2: $LSBs + OffetValue - Modulus$
The operation of Data-path2 can also be written as:
Datapath 2: $LSBs + OffetValue + \sim Modulus + 1$

The first three operands in Datapath 2 can be compressed into two data by one level CSA (Carry Save Adder)[3]. These two generated data can then be fed into a 32-bit adder. The carry in pin of this cascading adder, which is usually connected with 0, can be used to accept the remaining data 1. On the other hand, the carry out signal of the adder actually reflects the greatness of Intermediate and Modulus. That is, the signal is 1 if $Intermediate$ is less than $Modulus$, otherwise it is 0. Consequently, the carry out signal can be used as the select signal of the multiplexer so that the compare unit can be removed. The proposed modulo addressing unit structure is shown in Fig. 5(b). The critical path of this structure includes an inverter, a one level CSA, an adder and a multiplexer. Since one level CSA is actually an array of full adders, the delay of CSA equals that of a full adder. Further more, the delay on the inverter is negligible when calculating the entire delay of the timing path. Consequently, the delay on the critical path of this structure is significantly reduced compared with that of the former one. The implementation of the decrement modulo addressing follows the same idea as that of the increment modulo addressing. It can be achieved by adding control signals and multiplexers to the proposed structure so that both kinds of modulo addressing can be accomplished by sharing most of the components.

2.3 Fully Pipelined MAC Unit

In order to accelerate the operating speed of the DSP, MAC unit is divided into two pipe stages and takes a latency of two clock cycles to complete the operation. Instead of the conventional way to simply put multiplier and accumulator in different pipe stages[6], however, a different approach is employed in the division of the unit, as is shown in Fig. 6

In the first pipe stage, a radix-4 modified Booth encoder encodes the 32-bit multiplier and multiplicand into 17 partial products [4][5]. These partial products are compressed to two 64-bit data by Wallace Tree constructed with 4-2 compressors [4]. Traditionally these two results are put into an adder to produce the multiplication result. In this processor, however, the two results, together with the third operand which is sign-extended to 72 bits, are fed into a one level CSA, where they are compressed into another two intermediate results. In the second pipe stage, a multiplexer selects the signals piped down from the previous stage according to the SEL signal. If the SEL indicates a MAC or a multiplication operation, the multiplexer selects the signals from CSA. Otherwise, two 72-bit source operands are selected. A 72-bit adder then adds the selected signals to get the final result.

Since the adder inherent in a multiplier is substituted with a CSA, which has a delay of only one full adder, the proposed MAC structure achieves a more balanced pipeline than the traditional one. With the proposed structure, one MAC

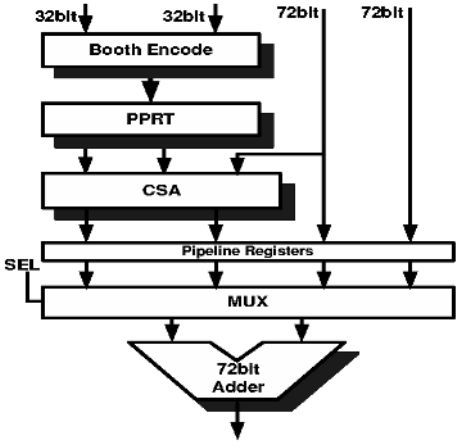


Fig. 6. The diagram of the proposed MAC unit, where PPRT stands for Partial Product Reduction Tree.

operation only needs one 72-bit adder. With the traditional style, however, two adders are involved in the MAC operation, one 64-bit adder for multiplication and one 72-bit adder for accumulation. Consequently, this structure can significantly reduce the area cost and the power consumption of the unit by the removal of 64-bit addition operation. Although the latency for a single multiply operation is increased, the fact that MAC is used more frequently than a single multiply operation in DSP algorithms justifies such kind of trade off.

2.4 Cache Strategy

The DSP3000 possesses a $2k \times 32$ bits instruction cache to enhance the performance and reduce power consumption by preventing the core from accessing external memory frequently. The cache is two-way set associative and adopts the write-through scheme. It also employs the LRU (Least Recently Used) algorithm to replace the data.

When cache miss occurs, the DSP core must enter wait state until the desired instructions are fetched from external program memory, which may take tens of clock cycles. In order to save power during that period of time, a clock gating approach is employed to hold the core state. Fig. 7 shows the clock gating circuit between the PLL and the DSP core. The $\overline{core\_hold}$ signal is latched by a latch, which is used to prevent glitches[7], and ANDed with the clock signal from PLL to generate the core clock. In the real implementation, the latched signal of $\overline{core\_hold}$ is ORed with scan_en before it is ANDed with the clock to meet the DFT(Design for Test) requirements. The scan_en signal is asserted only when the chip is in scan mode. When cache control module detects a cache miss, $\overline{core\_hold}$ is pulled down before the end of the clock cycle and the core clock is stopped. In that case, no operation would happen within the DSP core until the $\overline{core\_hold}$

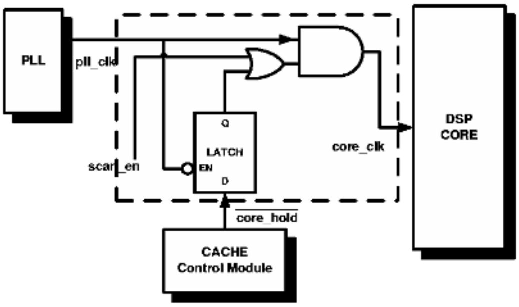


Fig. 7. Clock gating structure designed to stop the clock of the core when cache miss occurs

signal is released. With the proposed structure, the dynamic power consumption on the clock tree during cache miss is eliminated and thus the average power consumption on chip is further reduced.

3 Performance Analysis

The processor is modeled by Verilog-HDL and synthesized by Design Compiler with the SMIC (Semiconductor Manufacturing International Corporation) $0.18\mu m$ general standard cell libraries. The physical implementations of the processor, which include floorplanning, placement, clock tree synthesizing and routing, are carried out with Synopsys back end tools. Fig. 8 shows the final layout of DSP3000. The processor is fabricated with SMIC $0.18\mu m$ 1.8V 1P6M process and is tested on ADVANTEST T6672. The test result shows that processor can operate at a maximum speed of 300MHz with the average power consumption $30mw/100MHz$. Table 1 summarizes the main characteristics of the processor.

Table 1. DSP3000 Characteristics

Item	Characteristics
Process	SMIC 0.18um 1P6M
Core Voltage	1.8V
IO Voltage	3.3V
Core Size	$2.2mm \times 2.4mm$
Die Size	$4.8mm \times 4.8mm$
Operating frequency	$300MHz$
Average Power Consumption	$30mw/100MHz$

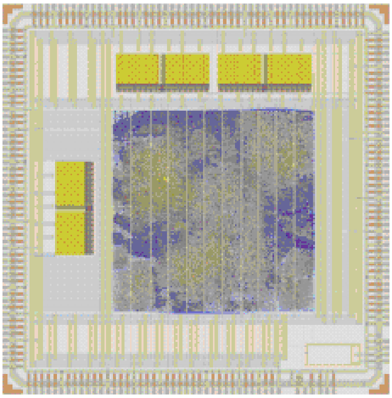


Fig. 8. The layout of DSP3000

Table 2. Comparison between DSP3000 and Other Commercialized DSPs

Item	DSP3000	TI C54X*	TI C55X*	Blackfin*
Category	32-bit fixed-point	16-bit fixed-point	16-bit fixed-point	dual MAC 16-bit fixed-point
Operating frequency	$300MHz$	$160MHz$	$300MHz$	$350MHz$
Benchmark:256-point FFT	2257 cycles	8542 cycles	4786 cycles	3176 cycles
Benchmark:Real FIR	$N/2+6$	$N/2+16$	$N/2+3$	$N/2+2$
Benchmark:Complex FIR	$4N+12$	$8N+13$	$2N+4$	$2N+2$
Benchmark:Delayed LMS	$2N+10$	$2N+14$	$2N+5$	$1.5N+4.5$

∗Source: http://www.ti.com ; http://www.analog.com

Table 2 makes a comparison between DSP3000 and some other commercialized DSPs. Several benchmarks are listed in the table, which include 256-point complex radix-2 FFT with bit reversal, real coefficient FIR, complex FIR and Delayed LMS(Least Mean Square) filter.All the benchmarks take the unit of clock cycles per output sample unless otherwise noted. The data of the commercialized DSPs are based on the benchmarks provided by the cited websites. Due to the optimized MAC unit and enhanced AGU capability, DSP3000 exhibits competitively low clock cycle latency and short execution time in running such programs.

4 Conclusion

This paper presents the overall architecture as well as the novel micro-architectures of DSP3000, which is a 32-bit fixed-point digital signal processor. With the improvement in pipeline organization, address generation and MAC operation, the processor enjoys high efficiency in executing DSP programs mean while achieves a low average power consumption. Test result shows it can reach an operating frequency of $300MHz$ with average power consumption $30mw/100MHz$. This processor can be applied in fields where high performance and low power consumption are both required, such as mobile phones and digital cameras.

Acknowledgement. This work was supported by National High Technology 863 Program under the grant of $No.2002AA1Z$.

References

1. Sanjive Agarwala,etc.: A 600MHz VLIW DSP. IEEE Journal of Solid-State Circuits, Vol.**37**,No.**11** (2002)
2. Farooqui A.A., Oklobdzija V.G: General Data-path Organization of a MAC Unit for VLSI Implementation of DSP Processors. Proceedings of the 1998 IEEE International Symposium on Circuits and Systems, Vol. **2** (1998) 260–263.
3. Jan M. Rabeay: Digital Integrate Circuits-A Design Perspective. Second Edition. Prentice Hall.(2003) 591–592
4. Norio Ohkubo, Makoto Suzuki, Toshinobu Shinobo,Toshiaki Yamanaka: A 4.4ns CMOS 54 × 54-b Multiplier Using Pass-Transistor Multiplier. IEEE Journal of Solid-state Circuits,Vol.**30**,No.**3** (1995) 1013–1015
5. Wen-Chang Yeh and Chein-Wei Jen: High-Speed Booth Encoded Parallel Multiplier Design. IEEE Trans. Computer.Vol. **49**,No.**7** (2000) 28–55
6. DSP56300 Family Manual-24-Bit Digital Signal Processor, Revision 3.0.Motorola Inc. (2000) 4-1–4-10.
7. Darren Jose: How to successfully Use Gated Clocking in an ASIC Design. SNUG. **33** (2002) 609–633

TengYue-1[1]: A High Performance Embedded SoC*

Lei Wang, Hong-yi Lu, Kui Dai, and Zhi-ying Wang

National University of Defense Technology, School of Computer,
Changsha, Hunan 410073, P. R. of China
wanglei@chiplight.com.cn

Abstract. TengYue-1 is a microprocessor subsystem for embedded applications. Its heart is a 32-bit RISC microprocessor based on an instruction set architecture (ISA) designed by us. Through a WISHBONE compatible on-chip bus, the microprocessor, a universal memory controller, a LCD controller and other peripheral I/Os formed the SOC. TengYue-1 has been implemented and verified in SMIC 0.18um CMOS technology, and the maximum clock frequency is 300MHz@1.8V. This paper presents the design and implementation of TengYue-1. We used 9 ARM benchmarks to evaluate the performance of the microprocessor and the results showed that it met our goal. We also found a simple solution to the memory access conflict problem caused by the microprocessor core and the LCD controller.

1 Introduction

Currently, embedded system is hailed by major semiconductors and mobile device manufacturers. They need simple, light, and low power micro-controller, not high performance general purpose microprocessor. Obviously, current design goal is lower power and higher performance within given constraints. And both on-chip and off-chip configuration of peripheral device must be possible for various market demands.

TengYue-1 is a design for embedded systems based on above characteristics. TengYue-1 is an 'island' containing a 32-bit RISC microprocessor core (named CH) and other macro modules, such as a universal memory controller, a LCD controller and several peripherals I/O devices. Hardware debugging support and a production test interface are also included. TengYue-1 was implemented in SMIC 0.18um CMOS technology. Maximum clock frequencies can be 300MHz@1.8v. The design is oriented for authentication and data encryption/decryption in information security application. Coprocessor interface and abundant peripherals make it easy for system integration.

The remainder of this paper organized as follows. Section 2 shows the architecture of TengYue-1, and section 3 describes the design of each component of TengYue-1 in

[1] TengYue: In Chinese means jump over.

* This work was sponsored by High-Tech Research and Development Program of China 863 (No. 2002AA1Z1080); NSFC (No.60173040)

P.-C. Yew and J. Xue (Eds.): ACSAC 2004, LNCS 3189, pp. 126–136, 2004.

detail. Implementation scheme is explained in section 4. Section 5 gives the result of performance evaluation. Finally, section 6 gives the conclusions.

2 Architecture of TengYue-1

As shown in Fig. 1, Tengyue-1 consists of a 32 bit RISC microprocessor core, a universal memory controller, a LCD controller, two UART serial ports, an I^{2}C interface, 32-bit GPIO interface, programmable interrupt controller, power management unit and test interface.

The microprocessor core is the heart of TengYue-1. The instruction set architecture is a typical RISC architecture. The instruction set is summarized in Table 1. At the design of the instruction set, we emphasize design for pipelining efficiency and efficiency as a compiler target. In contrast to ARM architecture [2], CH architecture is simple and efficient for implementation. After detailed analyzing of the execution of ARM instructions [3], on the base of ARM instruction set, we add shift instructions and reduce the addressing modes of data manipulation instructions. Thus the length of execution path is reduced. We increase the number of GPRs, to help the compiler to exploit more parallelism.

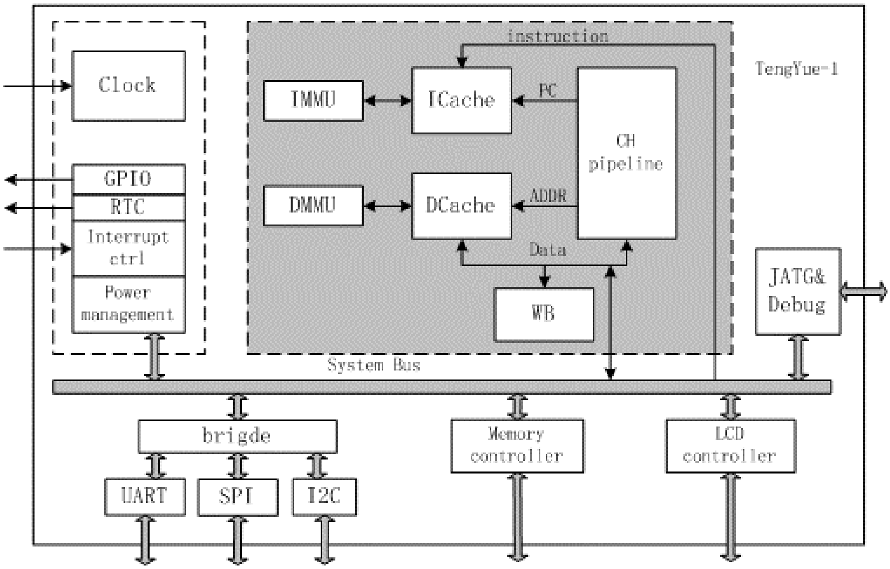


Fig. 1. Architecture of TengYue-1

CH has 32 32-bit general-purpose registers (GPRs), named R0, R1… R31. These GPRs can be mapped as two sets of 32-bit GPRs separately; each set contains 16 registers, for fast context switching in case of exception handling; or mapped as a single set of 32 32-bit GPRs. CH uses Harvard architecture, with separated instruction cache and data cache.

Table 1. Instruction set summary

Instruction Type	Number of this type
Branch	4 (16 conditions)
Data Manipulation	27
Load/Store	12 (6 addressing modes each)
Status Register Transfer	2
Exception generating Instructions	1
Coprocessor instructions	5 (2 for memory access, 6 addressing mode each)
JVM extension instructions	6
Total number	57

As shown in Fig. 1, the grey area is the RISC microprocessor core. To achieve high performance, we also have to pay attention to the clock frequency, execution efficiency (measured in terms of IPC or CPI), die size and power consumption in the design.

2.1 Microprocessor Core: CH

The 32-bit RISC microprocessor CH consists of two parts: the instruction pipeline and the memory subsystem. We will discuss these two parts in the following subsections.

2.1.1 Instruction Pipeline

In general, RISC architectures use pipelining as much as possible, in order to parallelize tasks and to use existing hardware resources efficiently. This usually results in a higher clock frequencies and therefore higher performance values. However the use of long instruction pipeline has some drawbacks. The longer the pipeline the more cycles is required to refill the instruction pipeline when executing a taken branch. CH used a classical five-stage pipeline, including Instruction Fetch stage (IF), Instruction Decode stage (ID), Execution stage (EXE), Memory Access stage (MEM) and Write Back stage (WB), as shown in Fig. 2. As CH has a single issue pipeline, the IF stage fetches one instruction from the instruction cache every cycle. The ID stage performs the instruction decoding and prepares the register operands. The instruction decoding can be done quickly because of the fixed instruction set encoding. Hazards (structure hazards and data hazards) detecting and resolving are also performed in ID stage. CH detects data hazards by Scoreboarding [1] and resolves them with bypassing and forwarding or simply stopping the successive instructions until the instruction that causes the hazards is completed. The EXE stage contains an ALU, a barrel shifter and a multiplier, executing arithmetic and logical instructions, shift instructions and multiplication instructions respectively. The EXE stage also generates the address for load/store instructions. Memory is accessed in MEM stage. In WB stage the results of the EXE stage or the data loaded from memory are written back to the register file.

Precise exception can be easily implemented since the hazards detecting and resolving algorithms are simple. When an interrupt or exception is detected, it only needs to flush the pipeline before handling the interrupt or exception.

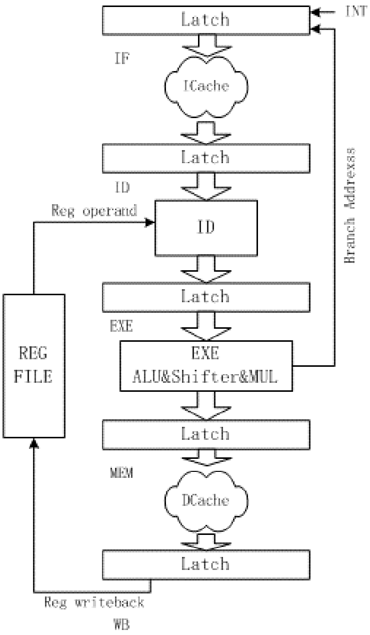


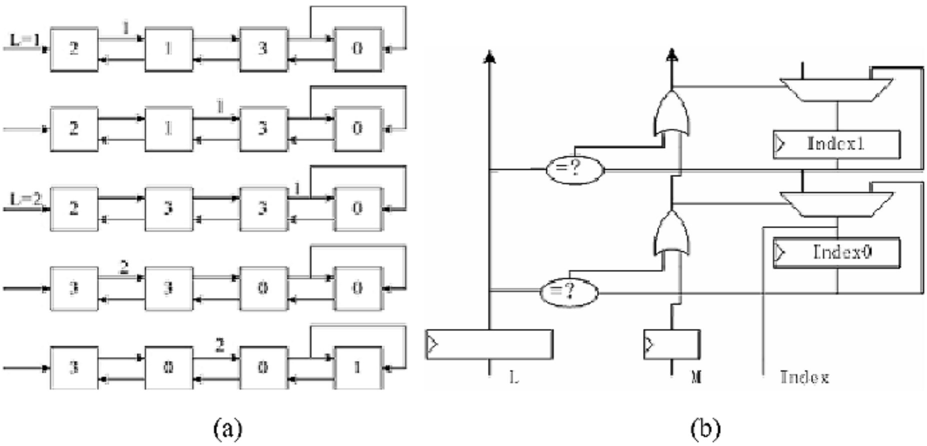
Fig. 2. Instruction Pipeline of CH

2.1.2 Memory Subsystem

The memory subsystem contains the instruction cache, data cache, write buffer and the memory management unit (MMU). The instruction cache and data cache are both 8KB, 4-way set-associative and both use LRU replacement algorithm. The write buffer has 32 16-byte entries. The instruction cache and data cache are both pipelined so that it is impossible to offer a instruction every cycle. All units in the memory subsystem can be enabled or disabled by user through configuring memory control register. Instruction TLB (Translation Look-aside Buffer) and data TLB are both full-associative.

The instruction cache uses virtual address as index and tag. A 7-bit identifier labels different processes. Thus the instruction cache does not need to be flushed on a context switch [1]. To avoid alias, the data cache uses physical address as index and tag. Using physical address has two advantages. Firstly, this resolves consistency problem of data cache; secondly, it is convenient for sharing data among processes (or threads). On a context switch, the data cache does not need to be flushed, so new processes can use the shared data without any costs [1].

Both the instruction cache and data cache use LRU replacement strategies. We implemented a simple systolic array that can keep track of LRU information for a set of cache lines. It can handle one cache access every cycle with less hardware cost [5]. The structure of a node of the array is shown in Fig.3 (b), where L is the LRU entry, M is the MRU entry, and index is the cache line access order kept by the array. The operation is illustrated in Fig.3 (a).



(a)

(b)

Fig. 3. (a) Operation of systolic array; (b) systolic array for implementing LRU algorithm

The write buffer works with the data cache. As the data cache uses write through policy. On write hit, the CPU write both the data cache and write buffer; On write miss, the CPU write into write buffer only. Whenever there is read miss, the write buffer is write back to the external memory before cache line refill begin. On read hit, the cache line in the data cache is the newest one, same as the external memory.

The virtual address is 32-bit and the physical address is 26-bit. Virtual memory is divided into pages. There are four kinds of page sizes: 1MB, 64KB, 4KB and 2KB. Memory protection information is kept with page table. The virtual-physical address translation is done by software. The TLB in the memory management unit acts as a cache to keep the historical conversion results, so a hit in TLB can accelerate the address translation.

2.2 Memory Controller

Interfaced with off-chip memory, the memory controller has 8 chip selects, each one is individually programmable. The memory types supported by TengYue-1's memory controller include SSRAM, SDRAM, FLASH and ROM etc. As a universal memory controller, the user can set the memory spaces mapped by the memory on each chip select and the timing sequences of the controller's interface by setting the chip select control registers and timing control registers to accommodate to different memory.

Fig. 4 shows the structure of the memory controller. The CPU bus interface and the memory interface are responsible for the communication with CPU and with memory individually. The configuration registers contain the information such as the types of memory, timing information, address mapping etc. All data paths and address paths are controlled by a FSM, which generates sequence of control signals according to the information stored in the configuration registers. The Power-on configuration block latches the value of the memory data bus during reset, which determines initial configuration of the memory controller and provides additional configuration bit for the system. The refresh counter and SDRAM control module are responsible for generating refresh cycles request and timing sequence for the attached SDRAM.

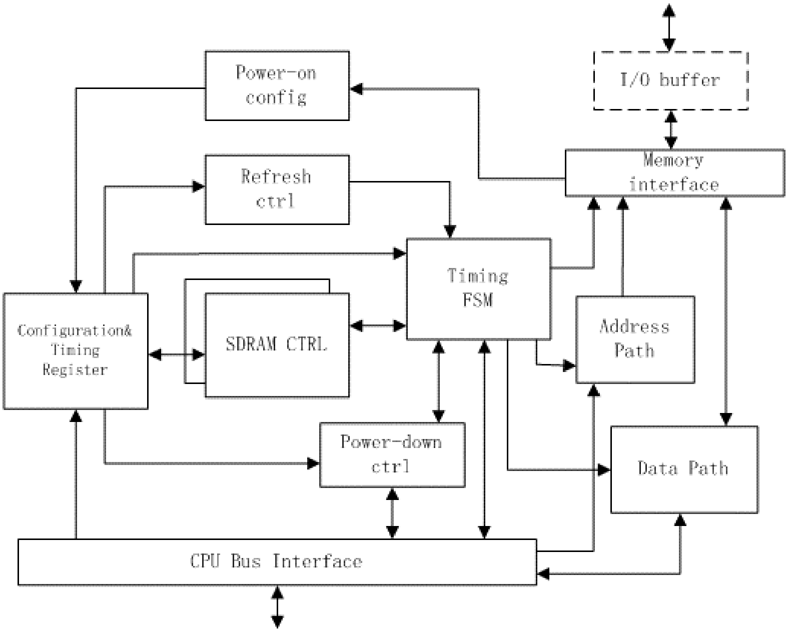


Fig. 4. Architecture of memory controller

The memory controller utilizes two clocks. One is the main clock of the system, the clock of microprocessor; the other is derived from main clock by dividing the main clock by two. The two clocks are required to be synchronized, but it's hard to keep strict synchronization in the actual physical circuits. We avoid the metastability problem caused by signals transfer across clock domains by two-level synchronization and this method works well in the implementation [4].

2.3 Peripherals

The peripherals of TengYue-1 include LCD controller, UART serial ports, SPI, I^{2}C, GPIO interfaces, and programmable interrupt controller, power management unit, hardware debug and test interface. All peripherals are connected to the microprocessor by the on-chip bus. The peripherals are mapped into the addressing space of main memory and accessed by the microprocessor through load/store instructions.

The LCD controller can provide independent horizontal/vertical synchronization and combinational synchronization output signals. The size of screen and the polarities of each video timing signal can be programmed by user, thus providing compatibility with almost all available LCD displays. The LCD controller can support a number of color modes, including 32bpp (bit per pixel), 24bpp, 16bpp, 8bpp grayscale and 8bpp pseudo color. The color lookup table is inside the controller, to reduce memory bandwidth request. Video memory and color table are both double bank, which can be used to reduce flicker and cluttered images. This feature is good for application such as video game and video stream application.

TengYue-1 has power manager. It has three modes: run mode, idle mode and sleep mode. These modes are used to reduce power consumption at times when some functions are not needed. When the chip is under the latter two modes, it can be awoken by interrupts or software conditions. The interrupt controller can connect up to 32 external interrupts. The interrupt priority levels and response modes (level sensitive or edge sensitive) can be programmed by the user. The hardware debug and test interface help the test of the chip and accelerate the development of application.

2.4 On-Chip Bus

The WISHBONE compatible on-chip bus of TengYue-1 connects multiple master devices to multiple slave devices. The microprocessor acts as the master device, while the memory controller and the various peripherals are slave devices. The LCD controller can access the memory directly through its own DMA channel, but it also works as the microprocessor's slave device. So it is not only a master device, but also a slave device. When a slave device has multiple master devices, their access sequence is decided by priority bit in the bus configuration register.

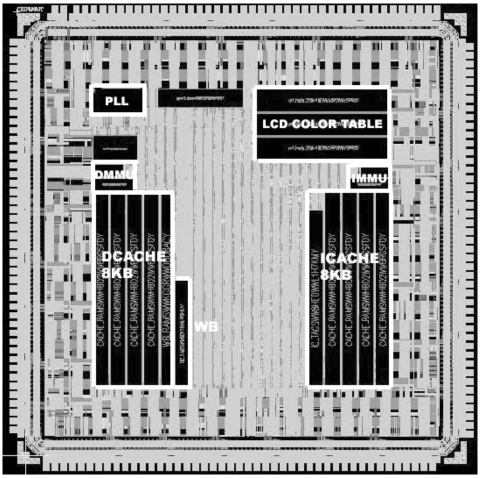


Fig. 5. TengYue-1 test chip's GDSII-based plot

3 Implementation

We use Synopsys Design Compiler for synthesis. The synthesis methodology is a combination of top-down and bottom-up synthesis. We use bottom-up synthesis for the critical modules by constraining their ports, paths, load and fan-outs; other modules are synthesized by top-down method. In the physical design, Physical Compiler is used to generate cell placement, freezing the data cache and instruction cache locations during the placement process and permitting the floorplanning to be data-driven. We generated the balanced clock tree with Cadence's CTGen. The maximum simulated skew across the core was 125ps under the worst case process and environmental conditions.

We fabricated the TengYue-1 in SMIC's 0.18um CMOS technology. Fig.5 shows GDSII-based plot of the test chip. Clock frequencies achieve 300MHz@1.8V; power dissipation is 0.47mW/MHz. Die size is 4.9mm*4.9mm in the total. For the microprocessor core, including instruction cache and data cache, die size is 4.73mm^2.

4 Performance Evaluation

4.1 Microprocessor Core Performance

Analysis of the results for 9 ARM benchmarks showed that the microprocessor core CH achieved an IPC count of about 0.64. All the benchmark programs are written in C language and compiled by gcc. Fig. 6 shows the CPI of each program.

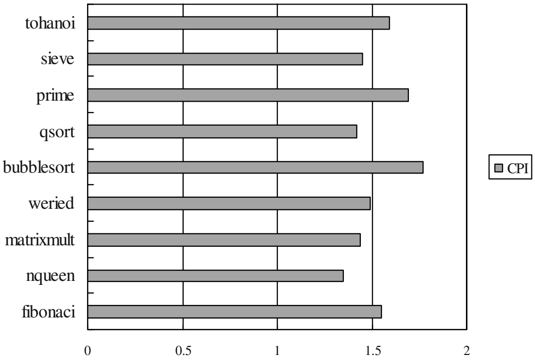


Fig. 6. IPC of benchmarks

4.2 Study of Memory Access Conflict Problem

Because the video memory of LCD is in the main memory, the memory controller has two master devices: the microprocessor and the LCD controller. Thus there exists conflict between memory access from the LCD controller and the microprocessor. If the LCD controller occupied most of the memory bus cycle, then the performance of CPU executing program or LCD driver maybe affected. For LCD driver, it can not write display information in time. One solution to this problem is to isolate the video memory of LCD from the main memory. The cost is adding another memory controller and the on-chip bus becomes more complex.

In our design, the microprocessor core and the LCD controller share the memory controller. To prove this method can satisfy the performance requirements of the system, we built a queue model to simulate this problem, as shown in Fig. 7.

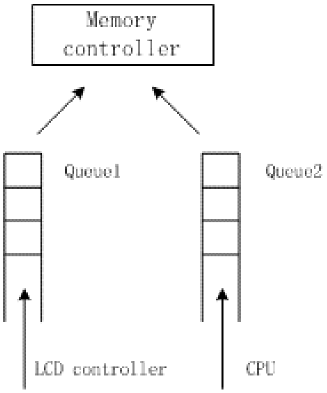


Fig. 7. Queue model

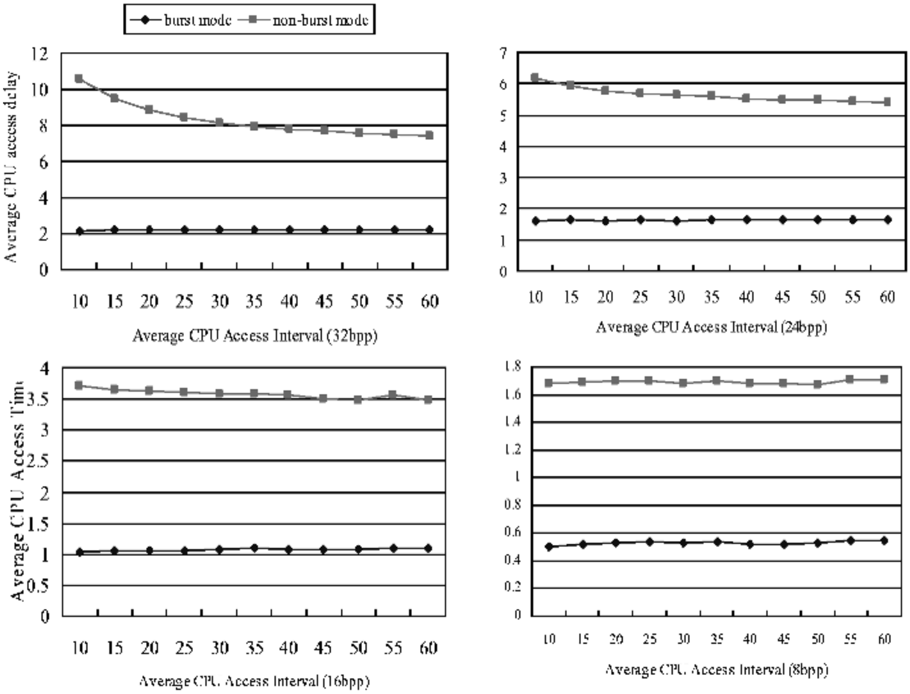


Fig. 8. Simulation result (CPU average memory access delay)

Queue 1 contains the memory access request from the LCD controller. As the LCD displays require pixel values being put on the screen in a fixed frequency, we suppose the arrival interval of video memory access request to be a fixed length distribution. Suppose the display period of the LCD selected is 156.25ns [7]. When the clock period of the microprocessor is 5ns(200MHz), to 32 bpp mode, a 32-bit word must be read out from memory every 31.25 cycles; to 16 bpp mode, a 32-bit word must be read out from memory every 62.5 cycles; 24 bpp and 8 bpp mode can be such deduced.

Queue 2 is for the microprocessor. We suppose that the arrival interval of memory access request from the microprocessor is an exponential distribution.

The memory controller acts as the server and the service time is a fixed value: 10 cycles without burst mode and 13 cycles with burst mode [6]. The burst mode returns four 32-bit words on one memory access while the non-burst mode returns just one. Suppose that the LCD controller has enough buffers to cache pixel data, the microprocessor doesn't use burst mode and the LCD has higher priority when conflict occurs.

We built a queue model by GPSS according to the previous assumptions and simulated the memory access behavior of the LCD controller and the microprocessor under different LCD display modes [8]. We studied the effect of memory access conflict to the performance of system. Fig.8 shows the simulation result.

From Fig.8 we can conclude that the LCD controller brings considerable impairment to the memory access of microprocessor when it doesn't use burst mode. In the worst case, about 33% load/store instructions of the microprocessor are delayed with an average of 11 cycles. However, when burst mode is enabled, the delay of memory access of the microprocessor is reduced to 1-2 cycles and the affected instructions are only 8% of all load/store instructions under the worst case. The cost is only three additional cycles to the memory access time of LCD controller.

Thus it can be seen that mapping the video memory to the main memory and sharing the memory controller between the LCD controller and the microprocessor can satisfy the performance requirement. The affection to the memory access of the microprocessor brought by the LCD controller can be reduced and even ignored when using the burst mode of the memory.

5 Conclusions

TengYue-1 is a high performance embedded SOC design. We studied the critical issues of the design and improved the overall performance by reducing the complexity of the hardware efficiently. This chip has been implemented and verified in SMIC 0.18um CMOS technology, and the frequency can achieve 300MHz@1.8V. TengYue-1 has a broad application prospect on RS encoding-decoding, information encryption/decryption and safety authentication.

References

1. Patterson D A, Hennessy J L. Computer Architecture: A Quantitative Approach. 2nd ed. San Francisco: Morgan Kaufman Publish, 1996.
2. S. B. Furber, *ARM System-on-chip Architecture*. Addison Wesley Longman(2000), ISBN: 0-201-67519-6.
3. ARM Inc. ARM Architecture Reference Manual [Z]. ARM DDI 0100D, Write Paper, 2000

4. Clifford E. Cummings, *Synthesis and Scripting Techniques for designing Multi-Asynchronous Clock Designs*, SNUG 2001 2.
5. J. P. Grossman, *A Systolic Array for Implementing LRU Replacement*, Project Aries Technical Memo
6. Micron Technology, *Synchronous DRAM Datasheet*
7. NEC, *TFT COLOR LCD MODULE datasheet*
8. Robert C. Crain, *SIMULATION WITH GPSS/H*, Proceedings of the 1998 Winter Simulation Conference

A Fault-Tolerant Single-Chip Multiprocessor

Wenbin Yao[1], Dongsheng Wang[2], and Weimin Zheng[1]

[1] Department of Computer Science and Technology, Tsinghua University, P. R. China
[2] Research Institute of Information Technology, Tsinghua University, P. R. China
{yao-wb, wds, zwm-cs}@tsinghua.edu.cn

Abstract. The microprocessor is a crucial component of a reliable system. With improvement in semiconductor manufacturing, more and more transistors may be integrated into a single chip with increased potential detriment to dependability. Fault-tolerant single-chip multiprocessors offer an ideal architecture for achieving high availability while maintaining high performance. The design of a fault-tolerant single-chip multiprocessor is described - from hardware redundancy to software support and firmware information strategies. The design aims at masking the influences of errors and automatically correcting system states, which differs from traditional approaches which mainly target errors in the memory and I/O subsystems. Dynamic recovery and reconfiguration are also described to provide adequate protection from catastrophic failure of the system.

1 Introduction

The growth of dependency on computer systems demands microprocessors which provide higher dependability whilst maintaining high computing performance. This is particularly true for mission-critical applications. With the development of deep-submicron technology, it is predicted that in the next 10 years a single chip can contain more than one billion transistors[1]. However, shrinking geometries, lower power voltages and high frequencies also have a negative impact on dependability.

Recently, as the trend toward thread-level parallelism matures, single-chip multiprocessors(CMP) present a promising solution to partly mitigate these influences[2-4]. CMPs integrating multiple processors into a single chip execute several threads concurrently to achieve high computing performance. The advantages of this technique include simplified design of critical paths and shrinking of development time and cost.

From an architectural viewpoint, CMPs combined with fault-tolerant techniques can further improve microprocessor dependability. Such designs follow two reliability principles. First, they observe the classical maxim: “simple is reliable”. A design achieves this goal by incorporating simple control logic and replacing the traditional complex parallel structures with multiple simple processors. Second, CMPs, which may use multiple identical processors for error detection and recovery, have inherent features leading to flexible fault-tolerant architectures. Different fault-tolerant strategies can be implemented neatly by reasonably dispatching available redundant components.

P.-C. Yew and J. Xue (Eds.): ACSAC 2004, LNCS 3189, pp. 137–145, 2004.

In this paper, we propose an architecture for a fault-tolerant single-chip multiprocessor (F-CMP). Differing from the IBM pSeries 690[5] which pursues RAS and can tolerate duration of repair, F-CMP provides high levels of reliability and availability with strong automatic recovery capability. The architecture is configurable and supports the replacement of faulty components and the degeneration to lower reliability levels when uncorrectable errors occur. Users can also specify a fault-tolerant mode corresponding to the dependability requirements of ant applications. Fault-tolerant strategies have been designed with special characteristics to tradeoff among hardware and software.

The rest of the paper is organized as follows. Section 2 presents an overview of F-CMP architecture. Section 3 discusses the fault-tolerant design techniques used in F-CMP, including hardware redundancy, firmware and software support. Section 4 describes the recovery strategies of F-CMP and Section 5 concludes.

2 System Overview

The F-CMP architecture is based on Tsinghua University's Thump-107. The Thump-107 is a RISC-based microprocessor targetting embedded applications. Its instruction set is a superset of MIPS-4Kc and is compatible with MIPS 32-bit RISC architecture. It has a 4k-byte instruction cache, a 4k-byte data cache and a 7-stage pipeline structure able to execute up to seven instructions per clock cycle.

From an architectural point of view, F-CMP is a closely coupled multiprocessor that contains four identical Thump-107 processors, a shared cache and necessary control logic needed to realize the fault-tolerant strategies. A logical overview is shown in Fig. 1.

F-CMP has six kinds of functional units:

- Four identical Thump-107 processors

The Thump-107 processor is an independent 32-bit CPU core, which implements the MIPS 4Kc instruction set plus four instructions for multimedia applications. In F-CMP, the processors reuse the basic Thump-107 structure, adding two instructions specially for implementing synchronization primitives used by a standard CMP. These two instructions are the load locked(LL) and store conditional(SC) instruction, respectively. Each processor also includes an 8k-bytes instruction cache, an 8k-bytes data cache and corresponding control logic. The L1 cache is a write-through primary cache that allows all processors to snoop on all writes performed.

- Fault Handling Mechanism

A fault handling mechanism consisting of a crossbar and four sets of fault-tolerant selectable logic was designed to detect computing errors by comparing results from independent processors. Logically, the crossbar controlled by the centralized arbitration controller connects the outputs of four identical processors with select logic. Four fault-tolerant computing modes are provided: one-mode, dual-mode, triple-mode and quadruple-mode redundancy, named after the numbers of participating processors.

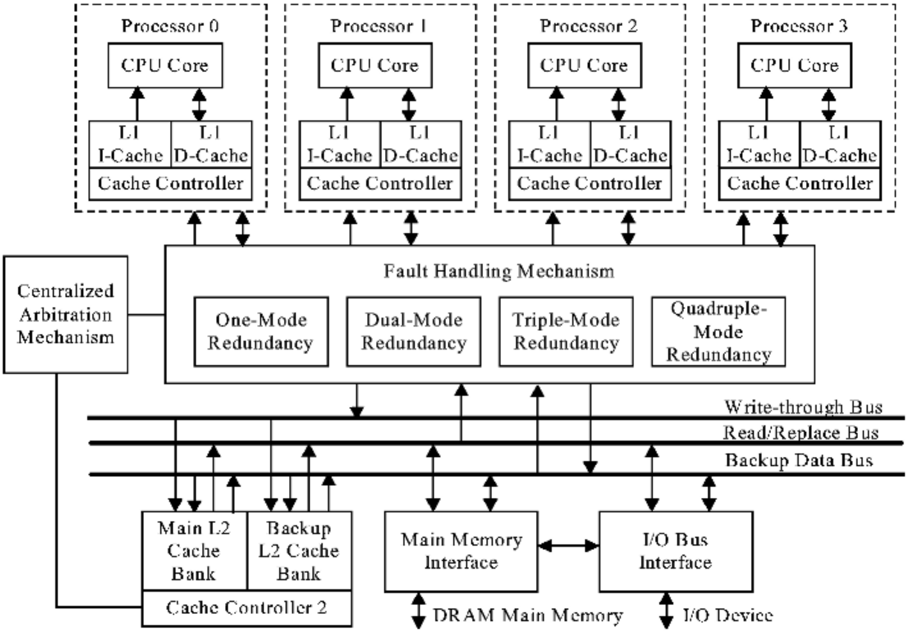


Fig. 1. Logical overview of F-CMP architecture

- A shared secondary cache

  Four processors share a unified L2 cache organized as two cache banks with separate controllers. The L2 cache can protect data with standard CRC words and its capacity is as large as 2M-bytes. In the non-fault-tolerant mode, the shared cache can be organized as a unified interleaved cache. In the fault-tolerant mode, the L2 cache consists of two independent sub-caches, each of which backs up the other. The main cache bank and the backup bank make up a fault-tolerant memory subsystem.

- A centralized arbitration controller

  Besides controlling the crossbar to provide fault-tolerance, the centralized arbitration controller manages access privileges on the shared bus. In F-CMP, three logic data buses including a write-through bus and a read/replace bus and a backup bus are used for data transmission. The backup bus is a standard data bus and may take over the tasks of the other buses if necessary.

- Main memory interface (MIU)

  The MIU handles all the interfacing transactions from and to F-CMP, including main memory accesses and external snoop processing.

- I/O interface unit

  The I/O interface unit handles all the input and output transactions of F-CMP.

3 Fault-Tolerant Design Techniques

F-CMP provides different levels of fault tolerance, including hardware redundancy, software support and firmware-based recovery. The architecture also allows system reconfiguration and dynamic graceful degeneration.

3.1 Fault-Tolerant Hardware Design

F-CMP provides some special structures to satisfy the requirements of different levels of fault tolerance. Here, three fault-tolerant strategies are presented:

- Fault-tolerant Computing

To achieve high reliability in the course of computing, three fault-tolerant strategies based on comparison techniques are provided in F-CMP. Fig. 2 shows the logic structure of different redundancy modes.

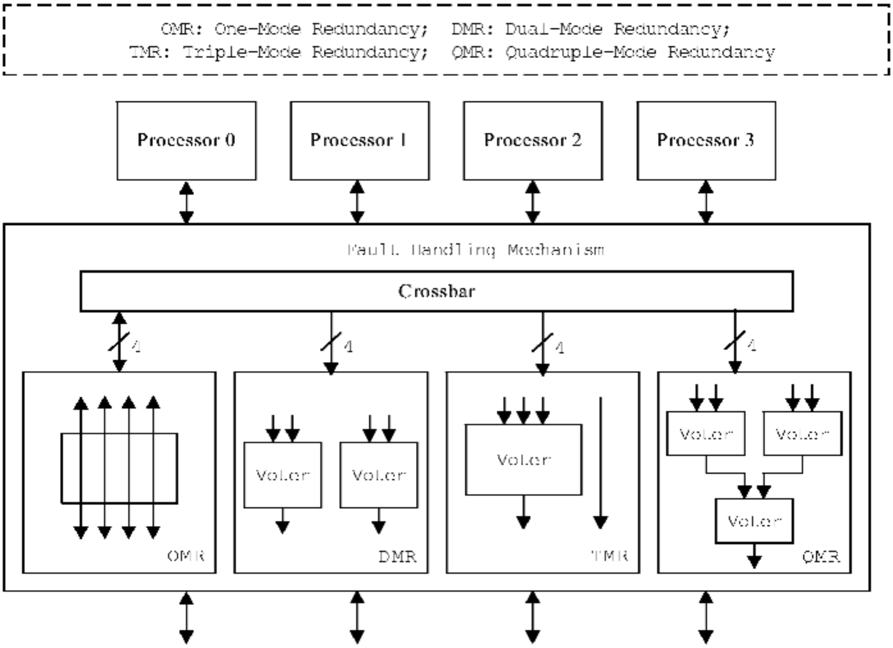


Fig. 2. Logical structure of fault handling mechanisms

In the fault handling mechanism, there are four sets of fault-tolerant logic. However, at any time, only one set of logic is activated.

One-mode redundancy represents a standard single-chip multiprocessor, in which every processor is an autonomous subsystem running applications independently. It is a non-fault-tolerant mode. In dual-mode redundancy, two processors form a simple comparison-based fault-tolerant subsystem while the others run as two independent single-processor subsystems. Similarly, triple-mode redundancy includes an autonomous single-processor subsystem and a voting fault-tolerant subsystem of three processors. The most complex form, quadruple-mode redundancy, uses all the four processors for fault-tolerant computing. In this mode, each pair of processors form a simple comparison-based fault-tolerant subsystem, and their outputs are compared once more to output a single result.

The fault-tolerant modes are controlled by an error-capture register (ECR). According to the different connection of the output of the processors, one-mode, dual-mode, triple-mode and quadruple-mode redundancy require 1 bit, 6 bits, 4 bits and 3 bits, respectively, in the ECR. As a result, the ECR has 14 controlling bits all together, each of which corresponds to a computing mode.

- Reliability Data Transmission

Connecting the processors, secondary cache and other control interfaces together are a read/replace bus, a write-through bus and a backup bus, along with address and control buses. While the read/replace and write-through buses are virtual buses, the physical wires are divided into multiple segments using repeaters and pipeline buffers. The backup bus is an independent physical bus, which implements standard reads and writes. Thus it can be used to replace the other two logical buses. As a result, the bus structure is actually a dual-bus system which protects transmitted information.

The read/replace bus acts as a general-purpose system bus for moving data between the processors, secondary cache and external interface to off-chip memory. The processors can fetch required data from secondary cache via the read bus simultaneously. Data from the external interface can be broadcast to both of the processors while it is sending to secondary cache.

The write-through bus permits F-CMP to use a write-update coherence protocol to maintain coherent primary caches. Data exchange among the processors is carried out via a write-through bus under the control of the centralized bus arbiter. When one processor modifies data shared by other processors, write broadcast over the bus updates all copies while the permanent machine state is written back to the secondary cache.

The backup bus design has two objectives. First, it can be used as an independent read and write bus to speed up communications between two levels of cache. Second, it can also be used as the single data bus when uncorrectable failures are detected on the other logical buses.

As in other high reliability microprocessor systems, data on the bus is augmented by a single-error-correct and double-error-detect Hamming ECC to further enhance fault-tolerance.

- Fault-tolerant Data Store

Both L1 and L2 caches have memory protection mechanisms designed to correct single event errors. Several bits are added to the banks to replace faulty bits. These techniques are analogous to the programmed steering logic in the structure of POWER4[6]. The controlling logic is activated by built-in self-test at power on.

As mentioned above, beside a normal unified cache, an L2 cache consisting of two banks can also implement a fault-tolerant memory subsystem by configuring one bank to back up the other. As a result, F-CMP may run in multiple different storage modes according the reliability states of caches. Possible storage configurations are shown in

Table 1, where '√' and '×' represent the availability and non-availability of the corresponding components, respectively.

Table 1. F-CMP Storage Configuration

Mode	L1 cache	L2 cache	
		Main Bank	Backup Bank
1	√	√	√
2	×	√	√
3	√	×	√
4	√	√	×
5	×	×	√
6	×	√	×
7	√	×	×
8	×	×	×

3.2 Firmware

Firmware was designed to record runtime error thresholds, indicating the number of corrected errors in the F-CMP microprocessor. The error information recorded in firmware becomes part of a system error log and is used for system reconfiguration. The replaceable components include the processors, cache banks and buses. When errors are detected in these components, error information is recorded in firmware.

Dynamic reconfiguration is implemented: firmware logs errors and is used to decide the current fault-tolerant mode. Hardware and firmware track periodically whether error numbers stay below a threshold. After exceeding this threshold, the system will initiate additional runtime availability actions, such as a controlled shutdown of processors and the replacement of a faulty cache line or a controlled shutdown of banks or even the whole L2 cache.

3.3 Software Support

Depending on hardware redundancy to achieve high reliability is not enough and software support plays a very important role on the design of F-CMP. Since each processor in F-CMP has its own program counter and register file, it is easy to execute multiple instruction streams in parallel. Actually, four independent instructions streams --- generally called threads --- can be simultaneously issued to the different processors to achieve software implemented fault tolerance (SIFT). The operations are scheduled under the control of the operating system.

As the mentioned above, the fault-tolerant modes of F-CMP are controlled by an error-capture register (ECR). This register is software-readable and can be reset by operating system. At bootstrap, the operating system may set the fault-tolerant mode of F-CMP according to application requirements and the current state of replaceable components in the microprocessor. Hardware and software may cooperate to implement runtime fault-tolerant handling, which is transparent to the users. F-CMP may also degenerate into a lower fault-tolerant mode if uncorrectable failures are encountered.

4 System Recovery Strategies

F-CMP implements concurrent error detection, dynamic fault isolation and error recovery while running. The most important principle of system recovery strategies is to guarantee the dependability of mission-critical applications even if system performance must be lowered. As a result, the replaceable components are considered first to take over from failed ones during the course of system reconfiguration. The error recovery flowchart of F-CMP is shown in Fig. 3.

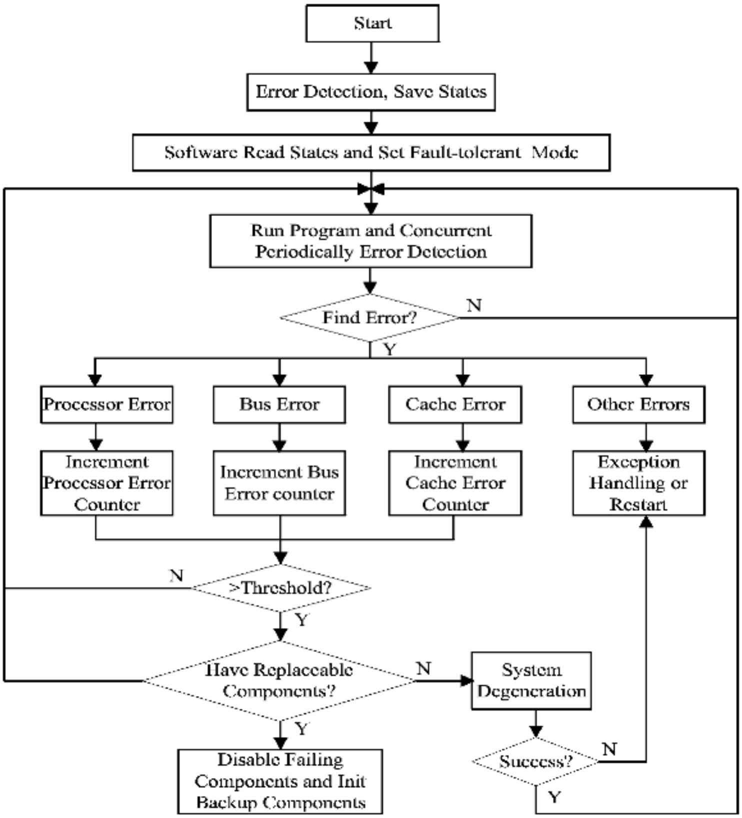

Start
Error Detection, Save States
Software Read States and Set Fault-tolerant Mode
Run Program and Concurrent Periodically Error Detection
Find Error?
N
Y
Processor Error
Bus Error
Cache Error
Other Errors
Increment Processor Error Counter
Increment Bus Error counter
Increment Cache Error Counter
Exception Handling or Restart
N
>Threshold?
Y
Have Replaceable Components?
N
System Degeneration
Y
Disable Failing Components and Init Backup Components
Success?
N
Y

Fig. 3. Error recovery flowchart of F-CMP

While the numbers of errors are beyond the capability of the current fault-tolerant mode, F-CMP may commence system degeneration. Corresponding to the fault-tolerant levels, there are three kinds of system degenerations in the level of processors and the buses and the caches. Processor-level degeneration has four forms: from quadruple-mode to dual-mode or one-mode redundancy, and from triple-mode to one-mode redundancy, and from dual-mode to one-mode redundancy, respectively. Bus-level degeneration may go from dual-mode to one-mode redundancy. In the cache-level degeneration, each cache line and bank and even whole caches may be removed as uncorrectable errors increase.

Besides the system degeneration, F-CMP has dynamic reconfiguration capabilities, viz. the state conversion between different fault-tolerant modes. Actually, the dual-mode and triple-mode and quadruple-mode redundancy have different dependability characteristics and complement each other. For applications requiring more processor power, dual-mode is suitable because the two groups of processors may execute different threads individually. But for other critical applications, triple-mode or quadruple-mode may be much better. System reconfiguration controlled by software includes two forms: conversion between dual-mode and triple-mode redundancy, and conversion between quadruple-mode and triple-mode redundancy.

5 Conclusion

We have described a fault-tolerant single-chip multiprocessor aimed at providing adequate protection from system failure. The design has configurable levels of hardware redundancy, software support and firmware information techniques. The architecture provides multiple fault-tolerant modes and is thus adaptable to different mission-critical applications: it also allows smooth conversion between the different modes. Since all the redundancy components and corresponding controllers exploit the “simple design” principle, it is easy to validate the reliability of subsystems and this leads to a high dependability system.

To verify the properties of the architecture, two related software tools have been written. One is a behavior simulator written in C, which performs functional validation. The simulator can simulate the operational behavior of F-CMP cycle-by-cycle. The other tool is a dynamic runtime error injection system, which randomly injects permanent and intermittent and transient errors into the system. Initial validation tests showed that the F-CMP architecture improved system dependability effectively for some mission-critical applications and reached the design target of initial dependability.

References

1. Cristian Constantinescu. Trends and Challenges in VLSI Circuit Reliability. IEEE Micro. Vol.23, No. 4(2003), 14-19
2. Sang-Won Lee, Yun-Seob Song, et al. Raptor: A Single Chip Multiprocessor. The First IEEE Asia Pacific Conference on ASIC. (1999) 217-220
3. Lucian Codrescu, D. Scott Wills, James Meindl. Architecture of Atlas Chip-Multiprocessor: Dynamically Parallelizing Irregular Applications. IEEE Transactions on Computers. Vol. 50, No. 1 (2001) 67-82
4. John Nickolls, L. J. Madar III. Calisto: A Low-Power Single-Chip Multiporcessor Communications Platform. IEEE Micro. Vol. 23, No. 4 (2003) 29-43

5. D. C. Bossen, A. Kitamorn, K. F. Reick, M. S. Floyd. Fault-tolerant Design of the IBM pSeries 690 System using POWER4 Processor Technology. IBM J. RES. & DEV. January Vol. 46, No. 1 (2002) 77-86
6. D. C. Bossen, J. M. Tendler, Kevin Reick. POWER4 System Design for High Reliability. IEEE Micro. Vol. 22, No. 2 (2002) 16-24

Initial Experiences with Dreamy Memory and the RAMpage Memory Hierarchy

Philip Machanick

School of ITEE, University of Queensland
Brisbane, Qld 4072, Australia
philip@itee.uq.edu.au

Abstract. This paper is a first look at the value of the RAMpage memory hierarchy to low-energy design. The approach used, *dreamy memory*, is to put DRAM in a low-power mode, unless it is referenced. Simulation results show that RAMpage provides a better overall speed-energy compromise than the conventional architecture used for comparison. The most energy-efficient RAMpage configuration in dreamy mode ran 3% faster and used 71% of the energy for DRAM of the best dreamy run of the conventional model. As compared with the best non-dreamy run time, the best dreamy time was 9% slower, but used under 17% of the energy for DRAM. The lowest-energy dreamy simulation used less than 16% of the DRAM energy of the fastest non-dreamy version, a very useful gain, given that DRAM uses significantly more power than the processor in a low-energy design. The most energy-efficient variant ran 12% slower than the fastest, allowing several trade-offs between speed and energy.

1 Introduction

The RAMpage memory hierarchy moves main memory up a level to replace the lowest-level cache with an SRAM main memory, while DRAM becomes a paging device. Previous work has shown RAMpage to be a potentially viable design in terms of hardware-software trade-offs [16] and that it scales better as the CPU-DRAM speed gap grows, particularly when taking context switches on misses to DRAM [14]. In this paper, the value of RAMpage in hiding DRAM latency is further explored by introducing the idea of *dreamy memory*.

Dreamy memory is kept in a low-power mode unless it is referenced. While waking the memory up incurs significant overhead, RAMpage could hide this overhead as has previously been demonstrated.

In desktop and server designs, with processor power consumption on the order of tens of watts or even over 100W, reducing memory power usage is not a major issue. However, with a low-energy design, DRAM energy usage becomes significant. A 128Mbyte DRAM as simulated in this study uses about 0.5W, as compared with a 500MHz processor of the ARM11 family [1], which uses about 0.2W. A small mobile device with a relatively modest memory therefore has to allocate a significant fraction of its energy budget to DRAM.

P.-C. Yew and J. Xue (Eds.): ACSAC 2004, LNCS 3189, pp. 146–159, 2004.

In this paper, the approach investigated is to use the self-refresh mode commonly available in double-data rate synchronous DRAM (DDR-SDRAM), which allows DRAM contents to be maintained with 1% of normal power [17], to implement dreamy memory. Simulations are based on parameters suited to a mobile device. The aim is to reduce DRAM energy usage to as close as possible to that of self-refresh mode, with performance as close as possible to that of full-power mode.

The remainder of this paper is structured as follows. Section 2 presents more detail of the RAMpage hierarchy and related research. Section 3 explains the experimental approach, while Section 4 presents experimental results. In conclusion, Section 5 summarizes the findings and outlines future work.

2 Background

2.1 Introduction

RAMpage was proposed [12] in response to the memory wall [21,9], which arises mainly with high-end systems, where processor improvements have not been matched by DRAM speed improvements. At the low end, energy use is a much more significant problem. RAMpage's ability to hide latency of (relatively) slow DRAM can potentially be used to hide the latency of waking a DRAM up from a low-power mode.

In this paper, low energy, rather than low power is the measure of interest, as we are concerned with total energy use over time, rather than an instantaneous measure.

The remainder of this section briefly surveys other approaches to low-energy memory design, followed by an outline of the RAMpage approach to the problem.

2.2 Low-Energy Memory Design

There have been several approaches to reducing the energy needs of memory.

IRAM (Intelligent RAM) was originally proposed to address the memory wall problem, by implementing a large DRAM on-chip with the processor, instead of the traditional trend of increasing on-chip cache size. While the on-chip DRAM is slower than an SRAM cache, it is faster than an off-chip DRAM [19]. More recently, IRAM has been shown to offer the potential for reduced energy usage, because of DRAM's lower energy requirement as compared to SRAM, and elimination of off-chip buses [6].

At the low end, work has been done on variations on memory organization like multiple banks (less commonly used banks can be put in low-power modes), finding optimum combinations of number of banks and bus width, and exploring compromises between performance-optimal and energy-optimal organization of caches and DRAM [2]. One specific proposal for a low-energy design for system-on-chip (SoC) applications is to organize static RAM into statically allocated banks, based on predicted data referencing behaviour [4]. The main problem with this approach is that it requires static allocation, and does not allow for changes in the relative sizes of the banks for different workloads.

The closest idea to that reported here are Power-Aware DRAM (PADRAM) [11] and Power-Aware Virtual Memory (PAVM) [8]: page placement is used in a memory in which different chips may be in different power modes. Frequently accessed pages are in a DRAM which is not in a low-power mode (or less often than other chips).

In a PADRAM study, it was shown that putting all DRAM into the lowest-power mode resulted in execution time of 2 to 60 times that of full-power mode, whereas a dynamic policy resulted in a relatively small speed loss, with significant energy saving. While various details of the PADRAM study differed from those reported in this paper (faster processor, smaller L2 cache, Rambus memory with higher wakeup latency), the most significant difference is that no operating system effects were modelled: single process execution times were reported, not a mix of workloads [11]. In addition, only a hardware-managed L2 cache was modelled, not a software-managed cache like the RAMpage SRAM main memory. RAMpage, especially with context switches on misses, relies on a multiprogramming workload to hide DRAM latency and is therfore able to get away with a simpler approach to managing DRAM.

PAVM has been investigated in more detail, but using an actual implementation on Linux and an otherwise-conventional memory hierarchy. Exploiting a combination of the different modes available in Rambus and dynamic page placement strategies, with DRAM energy savings of up to 59% with a heavy workload [8].

Since other low-energy techniques can apply to dreamy DRAM, approaches in areas such as reducing energy to drive a bus to DRAM [20] and reducing cache energy [10] have not been considered in detail as potential competing work.

2.3 The RAMpage Approach

RAMpage makes as few changes from a traditional hierarchy as possible. The lowest-level cache becomes the main memory (i.e., a paged virtually-addressed memory), with disk used as a secondary paging device. The RAMpage main memory page table is inverted, to minimize its size. Further, an inverted page table has another benefit: no TLB miss can result in a DRAM reference, unless the reference causing the TLB lookup is not in any of the SRAM layers [16].

RAMpage has in the past been shown to scale well in the face of the grown CPU-DRAM speed gap, particularly when context switches are taken on misses. The effect of taking context switches on misses is that, if other work is available for the CPU, waiting for DRAM can effectively be eliminated [14]. Performance characteristics of RAMpage have previously been reported [16, 13, 14]. For purposes of this paper, the key advantage of RAMpage is the ability to mask latency of DRAM references, with the aim of keeping DRAM in a low-power mode unless it is being referenced, without significant loss of speed.

Compared with most other approaches to low-energy memory systems, the RAMpage approach is very simple. No special hardware is needed, other than the RAMpage design itself. DRAM is put into a low-power mode, and turned on when it is referenced. As compared with the PADRAM approach, the architecture requires no complex dynamic placement strategy. Provided a process is ready to run on a miss to DRAM, the extra wake-up latency can be masked. PAVM is closer in philosophy, but RAMpage carries the idea further in managing the lowest-level cache in software, which has potential for other wins, as described in previous RAMpage work [16, 14].

The dynamic placement strategies of PADRAM and PAVM could be added to RAMpage, combining their benefits with a software-controlled SRAM main memory.

3 Experimental Approach

3.1 Introduction

This section outlines the approach to the reported experiments. Results are designed to be comparable to previously reported results as far as possible. The simulation strategy is explained, followed by some detail of simulation parameters; in conclusion, expected findings are discussed.

3.2 Simulation Strategy

The approach followed here is similar to that used in previously reported work. However, the processor speed characteristics are based on the ARM11 series [1] running at 500MHz. This processor consumes 0.2W at this speed; this power consumption makes the power needs of DRAM significant.

Simulations are trace-driven, and do not model the pipeline. It is assumed that pipeline timing is less significant than variations in DRAM referencing. Given that the ARM11 family only issues one instruction per clock and has accurate branch prediction, this approach to simulation is unlikely to introduce significant inaccuracies. For simplicity, the simulations do not use all features of the ARM11 series. The ARM11's two-level TLB is not simulated. Instead, a relatively small 1-level TLB is simulated. The RAMpage hierarchy is more disadvantaged by this approximation than a conventional hierarchy, since it relies on the TLB for mapping pages in the SRAM main memory, rather than in DRAM [15].

A standard 2-level hierarchy is compared to a similar version of a RAMpage hierarchy, with and without context switches on misses. RAMpage without context switches on misses is intended to convey the effects of adding associativity (with an operating system-style replacement strategy). Adding context switches on misses shows the value of having alternative work on a miss to DRAM. In all cases, the effect of running with DRAM permanently on is compared with the effect of running with DRAM in self-refresh mode, except when it is referenced.

In this study, given that energy and cost are more significant than for previous studies, L2 is reduced from 4 Mbytes to 1 Mbyte (the simulated 1MB SRAM consumes 0.8W [18]; 4 MB would use 3.2W, significant compared with a 0.2W processor). This reduction disadvantages RAMpage more than the standard hierarchy: part of the SRAM main memory is reserved for operating system data and code: in addition to a page table, RAMpage reserves 32 Kbytes for the operating system. The ARM11 series includes 64K bytes of SRAM (tightly coupled memory, TCM) which could be used for the operating system in RAMpage; the page table would also fit for SRAM page sizes of 256 bytes or more. This option was not explored in this study; using TCM, which operates at cache speed, in RAMpage simulations would likely result in significant speed gains.

3.3 Simulation Parameters

The processor modelled in this paper is slower than in recent RAMpage work, if comparable to one of the speeds in recent older work [16], to take into account the slower speeds of low-energy designs.

A major difference from previously reported results, which used Direct Rambus, is use of double-data-rate synchronous DRAM. The DDR-SDRAM modelled [17] has average power usage of 200mA, and self-refresh mode which uses 2mA, both at 2.6V. In self-refresh mode, the external clock is turned off, and contents of DRAM is maintained without external intervention. Actual DRAM power usage varies according to the reference pattern, but for this preliminary work, an average value is used, and the same value is used for entry to and exit from self-refresh mode. In previous work, detail of the DRAM was not considered important, as fixing DRAM speed while speeding up the CPU represented the increasing CPU-DRAM speed gap. In this paper, DRAM detail is more important because power usage is timing-dependent.

The following parameters are similar to previous simulations except as noted, and are common across RAMpage and the conventional hierarchy:

- L1 cache – 16 Kbytes each of data and instruction cache, physically tagged and indexed, direct-mapped, 32-byte blocks, 1-cycle read hit time, 12-cycle penalty for misses to L2 (or RAMpage SRAM main memory)
- TLB – 64 entries, fully associative, random replacement, 1-cycle hit time, misses modelled by interleaving a trace of page look-up software
- DRAM – DDR400 SDRAM: 40ns before first reference starts, 64-bit 5ns bus (data moves every 2.5ns: transfer rate approximately 0.3ns per byte; DRAM time to exit self-refresh is 1μs, and time to enter self-refresh mode is 20ns)
- paging of DRAM – inverted page table: same organization as RAMpage main memory for simplicity, the workload is preloaded, so there are no page faults to disk; for energy calculations, a 128MB DRAM is assumed
- TLB and L1 data hits are fully pipelined: they do not add to execution time; only instruction fetch bits add to simulated run time; time for replacements or maintaining inclusion are costed as L1d or TLB "hits"

A context switch (modelled by interleaving a trace of text-book code) is generally taken every 500,000 references, though RAMpage with context switches on misses also switches processes on a miss to DRAM. TLB misses are handled by inserting a trace of page table lookup code, with variations on time for a lookup based on probable variations in probes into an inverted page table [16].

Specific to conventional hierarchy. L2 cache is 2-way associative, 1Mbyte. The bus connecting L2 to the CPU is 128 bits wide and runs at one third of the CPU issue rate (6*ns* versus the CPU's 2 *ns*). The miss penalty from L1 to L2 overall is 12 CPU cycles. Inclusion between L1 and L2 is maintained [7], so L1 is always a subset of L2, except that some blocks in L1 may be dirty with respect to L2 (writebacks occur on replacement).

The TLB caches translations from virtual pages to DRAM physical frames.

Specific to RAMpage hierarchy. The TLB maps the SRAM main memory. Full associativity is implemented by a software miss handler. The operating system takes up 9 SRAM main memory pages when simulating a 4 Kbyte-SRAM page (36 Kbytes), up to 752 pages for a 128 byte block size (94 Kbytes).

The SRAM main memory uses an inverted page table. TLB misses do not reference DRAM, if the original reference can be found in an SRAM level.

Inputs and variations. Traces used are from the Tracebase trace archive at New Mexico State University[1]. Although these traces are from the obsolete SPEC92 benchmarks, they are sufficient to warm up the size of cache used here, because 1.1-billion references are used, with traces interleaved to create the effect of a multiprogramming workload.

To measure variations on energy use, the size of the SRAM main memory page (or L2 block size in the conventional model) was varied from 128 bytes to 4 Kbytes, and the simulation was instrumented to track energy use.

In dreamy mode, it was assumed that if a DRAM access started before the previous one had completed, DRAM would still be awake. Otherwise, once a DRAM reference completed, it was put into self-refresh mode. For comparison, simulations were run with DRAM permanently in full power mode. The simulator allows for a lag after references before entering self-refresh mode, but this option is still to be explored.

Total energy was calculated by multiplying time in each mode by the power of that mode.

3.4 Expected Findings

It was expected that speed differences would not be the most significant finding, given that early studies [16, 13] showed little difference between RAMpage and the conventional model at the clock speed being modelled in this paper. It was expected that the introduction of dreamy mode would have less of an effect on

[1] See `ftp://tracebase.nmsu.edu/pub/traces/uni/r2000/utilities/` and `ftp://tracebase.nmsu.edu/pub/traces/uni/r2000/SPEC92/`.

RAMpage than on the conventional model, given that RAMpage has been shown to be more tolerant of an increased DRAM latency, especially when context switches are taken on misses [14].

With a significantly smaller SRAM main memory than in earlier experiments, it was expected that RAMpage, which pins parts of the operating system in the SRAM main memory, would be less competitive on speed than in earlier experiments even in dreamy mode, where the increased effective DRAM latency would make this experiment closer to earlier ones with faster processors.

RAMpage, however, has the potential to show a better overall combination of not only lower speed loss in dreamy mode and lower overall energy use in dreamy mode, than the conventional hierarchy. Since RAMpage in general (and more specifically when contect switches are taken on misses) spends a lower fraction of its time waiting for DRAM, it is likely that it will need DRAM to be in full-power mode less often than the standard hierarchy does.

4 Results

4.1 Introduction

This section presents results of simulations, with some discussion of their significance. The main focus here is on comparing the effects of varying the memory hierarchy on energy and power use.

Figure 1 shows an overall comparison of all the variations measured. Of most interest is the fact that it's hard to tell apart speed variations of the best cases for each configuration on the same scale, whereas energy variations for dreamy and non-dreamy cases are clearly separated. This observation illustrates that aiming to save energy while minimizing performance loss is achievable.

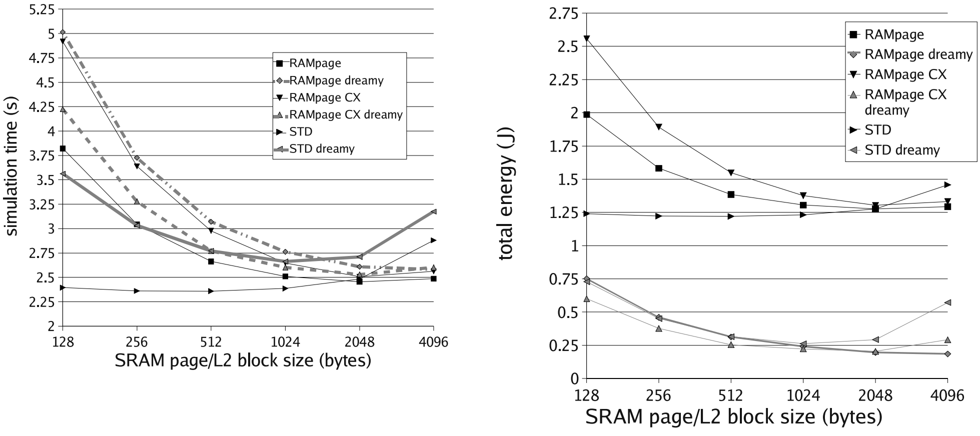
The remainder of this section presents more detail of results. Speed variations are followed by energy variations. Finally, design trade-offs are considered.

4.2 Speed Variations

Speed variations are shown in Table 1. Speedups are shown for the non-dreamy case of the best measured time versus each other time. For dreamy times, speedups are given both relative to the same parameters with and without dreamy mode (2nd-last column) and the best non-dreamy time (last column). The best dreamy and non-dreamy times are highlighted.

The best dreamy run time is for RAMpage with context switches on misses, with a 2KB SRAM page size. Execution time here is 9% slower than for the best non-dreamy case (conventional hierarchy, 512B L2 block size). More speed variation is accounted for by variations in the SRAM page or L2 block size than by using or not using dreamy mode. The slowest dreamy simulated execution time is 5.06s; the slowest non-dreamy time is 4.92s, a difference of under 3%.

The standard hierarchy's best dreamy time (1KB L2 block size) is 15% slower than the best non-dreamy time, while the best dreamy RAMpage time without context switches on misses is 12% slower than the best non-dreamy time.



(a) Overall Speed Comparison

(b) Overall DRAM Energy Comparison

Fig. 1. Comparison of speed and energy usage.In all figures, "CX" means with context switches on misses.

Table 1. Speed variations. Each row shows standard hierarchy (top), RAMpage without (middle) and with context switches on misses (bottom).

L2 block/	Non-Dreamy		Dreamy Times (s)			Dreamy Speedups	
page size	time (s)	best speedup	asleep	awake	total	non-dreamy	best
128	2.395	1.015	2.224	1.378	3.602	1.504	1.526
	3.821	1.619	3.654	1.409	5.062	1.325	2.145
	4.919	2.085	3.127	1.124	4.251	0.864	1.802
256	2.362	1.001	2.220	0.845	3.065	1.298	1.299
	3.043	1.290	2.910	0.856	3.766	1.238	1.596
	3.638	1.542	2.602	0.698	3.301	0.907	1.399
512	2.359	1.000	2.220	0.584	2.804	1.188	1.188
	2.663	1.129	2.534	0.577	3.111	1.168	1.318
	2.977	1.262	2.328	0.466	2.794	0.938	1.184
1024	2.386	1.011	2.224	0.481	2.705	1.133	1.146
	2.510	1.064	2.368	0.439	2.807	1.118	1.190
	2.647	1.122	2.229	0.406	2.635	0.996	1.117
2048	2.483	1.052	2.237	0.540	2.777	1.119	1.177
	2.454	1.040	2.304	0.354	2.658	1.083	1.127
	2.506	1.062	2.201	0.371	2.572	1.026	1.090
4096	2.879	1.220	2.293	1.077	3.371	1.171	1.429
	2.487	1.054	2.304	0.334	2.638	1.061	1.118
	2.562	1.086	2.139	0.538	2.677	1.045	1.135

While RAMpage doesn't do well with small SRAM page sizes – as reported in earlier work [16] – time variations for cases with reasonable SRAM page sizes are low considering the relatively large energy saving of dreamy mode. RAMpage with and without context switches on misses does not differ as significantly as in earlier studies with a large CPU-DRAM speed gap and large L2 [14]. Dreamy mode does increase the effective CPU-DRAM speed gap: an extra miss penalty of 500 clock cycles similar to increasing the processor speed to the speeds previously modelled. However, as expected, the smaller SRAM main memory used in this study disadvantages RAMpage more than the conventional model.

More data is needed to understand why RAMpage with context switches on misses is faster in some cases in dreamy mode. A possible explanation is that dreamy mode, with its longer latency for DRAM accesses, loses less performance to context switches before a working set has loaded fully into SRAM.

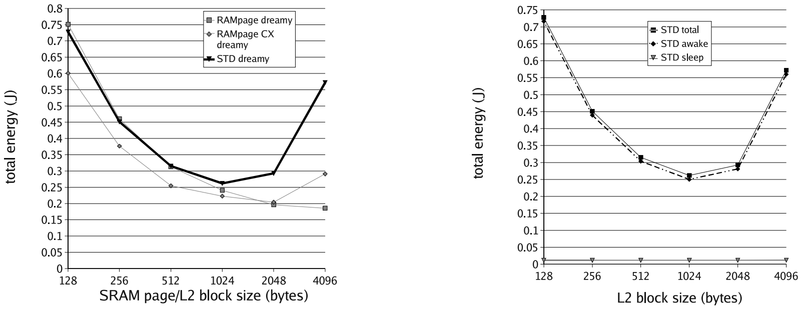
4.3 Energy Variations

Table 2 shows the simulated DRAM energy usage for each variation.

The lowest-energy non-dreamy case is the standard hierarchy with a 512B L2 block size, which also has the quickest execution time. For dreamy runs, however, the lowest-energy case is RAMpage without context switches on misses for a 4KB SRAM page size, as compared with the fastest case: 2KB page size, *with* context switches on misses. The reason for this discrepancy results from the fact that in

Table 2. DRAM energy use. Each row shows standard hierarchy (top), RAMpage without (middle) and with context switches on misses (bottom).

L2 block/	Non-Dreamy		Dreamy Energy				
page size	energy (J)	× best	asleep (J)	awake (J)	total (J)	% full	× best
128	1.241	6.7	0.0116	0.716	0.728	58.7	3.92
	1.987	10.7	0.0190	0.732	0.751	36.9	4.05
	2.558	13.8	0.0163	0.584	0.601	25.8	3.24
256	1.223	6.6	0.0115	0.440	0.451	22.9	2.43
	1.582	8.5	0.0151	0.445	0.460	39.2	2.48
	1.892	10.2	0.0135	0.363	0.377	21.3	2.03
512	$\boxed{1.220}$	6.6	0.0115	0.304	0.315	39.2	1.70
	1.385	7.5	0.0132	0.300	0.313	21.3	1.69
	1.548	8.3	0.0121	0.242	0.255	22.9	1.37
1024	1.232	6.6	0.0116	0.250	0.262	21.3	1.41
	1.305	7.0	0.0123	0.228	0.241	22.9	1.30
	1.376	7.4	0.0116	0.211	0.223	39.2	1.20
2048	1.276	6.9	0.0116	0.281	0.293	22.9	1.58
	1.276	6.9	0.0120	0.184	0.196	20.1	1.06
	1.303	7.0	0.0114	0.193	0.204	44.2	1.10
4096	1.458	7.9	0.0119	0.560	0.572	39.2	3.08
	1.293	7.0	0.0120	0.174	$\boxed{0.186}$	14.4	1.00
	1.332	7.2	0.0111	0.280	0.291	21.9	1.57



(a) Overall Dreamy Energy Comparison

(b) Standard Energy Breakdown

Fig. 2. All *vs.* Standard dreamy energy usage.

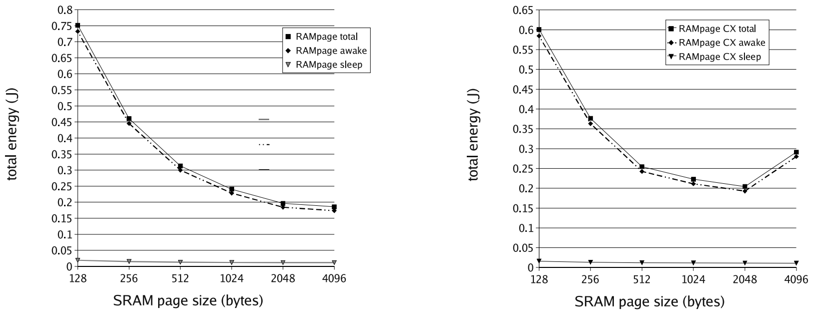
the 4KB case, DRAM is awake for a smaller total time (lower "awake" energy). The time DRAM is awake depends on time that transfers take. A larger page size may reduce the miss rate, but total transfer time may increase. Context switches on misses hides this effect by doing other work on a miss. However, increased energy is not disguised by overlapping transfers with other work.

Figure 2 compares energy use in dreamy mode for all variations with a breakdown of energy use by the standard architecture. Energy use increases significantly for large cache block sizes in the standard architecture, which is less true of RAMpage variations. The reason for this behaviour of the standard model can be seen in Figure 2. As L2 block size increases, energy use while asleep decreases, but awake energy increases, corresponding to a larger fraction of time being spent waiting for DRAM (as confirmed by the increase in execution time for the standard dreamy simulations for larger L2 block sizes, in Figure 1(a)).

Figure 3 compares RAMpage variations. The increase in energy use for context switches on misses with a 4KB page size needs further investigation. A likely cause is increased contention for SRAM pages resulting from the higher context switch rate. Large pages (or cache blocks) are likely to have the highest benefit if their prefetch effect can be put to good use.

4.4 Overall Trade-Offs

In summary, the fastest time does not necessarily correspond to lowest energy use, even if the system is not operating for as long overall. The quickest dreamy run time was 2.57*s* (context switches on misses, 2KB SRAM page size), while the lowest-energy variant took 2.64*s* (4KB SRAM page size, no context switches on misses). The lowest-energy variant ran about 3% slower than the fastest dreamy variant, or 12% slower than the fastest non-dreamy variant).



(a) No Context Switches on Misses

(b) Context Switches on Misses

Fig. 3. Comparison of RAMpage dreamy energy breakdown.

Electing to run RAMpage in its fastest dreamy mode would require 10% more energy for DRAM than its fastest variant. However, the overall fastest version (conventional, 512B L2 blocks) needs 6.6 times the energy of the most energy-efficient version, or 6 times the energy of the fastest dreamy variation.

A designer therefore can balance choices between maximum speed (no dreamy mode, standard two-level cache) and maximum energy saving (RAMpage without context switches on misses, SRAM page size chosen for lowest energy). As a compromise, it would be possible to use RAMpage with context switches on misses, with sub-optimal energy use, but better performance.

These speed-energy trade-offs only represent DRAM energy. The CPU and SRAM modelled use 1W (0.2W and 0.8W, respectively). For a run time of 2.57s, the pair uses 2.57J whereas over the best run time of 2.36s, the total goes to 2.36J. This difference is easily justified by saving over 1J in DRAM energy but, nonetheless, a more comprehensive energy analysis of the whole system is needed. For example, PADRAM runs uses *more* energy in its equivalent of a simply dreamy mode than without [11], probably because of its relatively small L2 cache. The relatively large L2 used here, on the other hand, uses more energy than one would like for a low-energy design.

5 Conclusion

5.1 Introduction

This paper has presented an initial study of use of the RAMpage memory hierarchy to reduce DRAM energy usage. The approach used was to simulate a *dreamy* memory, in which DRAM is turned off except when referenced. The motivation for this study is previous results which have showed RAMpage to be more tolerant of increased DRAM latency than a conventional hierarchy.

The remainder of this section summarizes results, outlines future work and presents overall conclusions.

5.2 Summary of Results

RAMpage, with the option of context switches on misses, presents some useful trade-offs in choosing an energy-speed design trade-off. Assuming a relatively low-energy processor design (as well as low-energy components for the remainder of the system), dreamy energy savings could be significant. The fastest configuration uses almost 7 times the energy of the most energy-efficient one, for a performance gain of only 12%. The performance cost of dreamy mode can be brought down to 9% by a relatively modest compromise on energy saving: this dreamy configuration still uses a sixth of the energy of the fastest version.

The best overall compromise is achieved by RAMpage with context switches on misses, though by a less significant margin than in earlier studies, which showed this variant to be most tolerant of high DRAM latencies [14]. A relatively small SRAM layer makes RAMpage less competitive than in these earlier studies.

5.3 Future Work

It is important that, while the savings were achieved with modest speed loss, overall energy usage should take into account other parts of the system, which would use more energy if left in full power mode for a longer time. If energy for the processor and L2 are added in, the fastest dreamy version also has the lowest overall energy by a small margin 2.78J versus 2.82J for the version with lowest DRAM energy). This should be compared against 3.58J for the best non-dreamy version, a saving of 29% which is useful but not as dramatic as a factor of 6.

Results should be extended to a more detailed analysis of overall system energy, including low-energy variations on caches, and low-energy versions of faster processors. The simulated SRAM has a relatively low latency for waking up from low-power mode (50% of the latency of an L2 hit). Since 1MB SRAM (0.8W) uses more power than DRAM in full-power mode (0.52W) – as simulated here – this would be a useful variation to explore.

A RAMpage implementation on the L4 Pistachio kernel [5] is planned. This kernel is small enough to permit implementation of its minimum memory-resident data and code in the 64KB static RAM memory in the ARM11 family. Using this extra SRAM would also make it viable to implement RAMpage with a smaller SRAM main memory, a significant factor in the overall energy budget of this kind of system. L4 has been ported to the M5 architecture simulator [3] by the NICTA group at University of New South Wales, creating the possibility of RAMpage on a full-system simulator, a goal of earlier work.

The existing simulator will be used to experiment with further variations on energy-efficient memories. For example, instead of an SRAM main memory, main memory could be implemented in a small fast permanently powered up DRAM, with the remaining DRAM operating as a dreamy paging device.

5.4 Overall Conclusion

In this latest study, investigating dreamy memory model has shown the potential for RAMpage in low-energy designs. While RAMpage did not run in the shortest time in full-power mode (expected with a relatively slow processor), it did have both the fastest and lowest-energy measurements in dreamy mode.

Results showed a fair fraction of the potential energy gain: full power mode needed 100 times that of self-refresh mode; the lowest-energy case used less than a sixth of full DRAM power. The aim of achieving close to an average of self-refresh power use with as close as possible to full speed has been partially met. More sophisticated approaches to minimizing power use (e.g., keeping power on for a period after a reference, exploiting a wider range of low-power modes, and dynamic page placement policies, as in PAVM) could further reduce energy use.

The design trade-offs discussed here represent a starting point: overall low-energy system design requires design of the whole system to minimise energy use. Just as Amdahl's Law shows that focus on one area of speed improvement has diminishing returns, we need to be careful not to interpret energy savings in isolation. Nonetheless RAMpage shows promise in the area of low-energy design, and this study will be followed up with others.

Acknowledgements. Financial support for this work has been received from the University of Queensland. I would like to thank Gernot Heiser for proposing that I investigate energy management using RAMpage.

References

1. ARM. *The ARM11 Microprocessor and ARM PrimeXsys Platform.* ARM, October 2002. `http://www.arm.com/pdfs/ARM11%20Core%20&%20Platform%20Whitepaper.pdf`.
2. Luca Benini, Alberto Macii, and Massimo Poncino. From Energy-aware design of embedded memories: A survey of technologies, architectures, and optimization techniques. From *ACM Trans. on Embedded Computing Sys.*, 2(1):5–32, 2003.
3. N. L. Binkert, E. G. Hallnor, and S. K. Reinhardt. From Network-oriented full-system simulation using M5. From In *Sixth Workshop on Computer Architecture Evaluation using Commercial Workloads (CAECW)*, pages 36–43, February 2003.
4. Yun Cao, Hiroyuki Tomiyama, Takanori Okuma, and Hiroto Yasuura. From Data memory design considering effective bitwidth for low-energy embedded systems. From In *Proc. 15th Int. Symp. on System Synthesis*, pages 201–206, Kyoto, Japan, 2002.
5. Uwe Dannowski, Kevin Elphinstone, Jochen Liedtke, Gerd Liefländer, Espen Skoglund, Volkmar Uhlig, Christian, Ceelen Andreas, and Haeberlen Marcus Völp. From The L4Ka vision. From Technical report, University of Karlsruhe, System Architecture Group, April 2001. From `http://i30www.ira.uka.de/research/documents/l4ka/L4Ka.pdf`.
6. Richard Fromm, Stylianos Perissakis, Neal Cardwell, Christoforos Kozyrakis, Bruce McGaughy, David Patterson, Tom Anderson, and Katherine Yelick. From The energy efficiency of IRAM architectures. From In *Proc. 24th Int. Symp. on Computer Architecture*, pages 327–337, Denver, CO, 1997.

7. J.L. Hennessy and D.A. Patterson. From *Computer Architecture: A Quantitative Approach*. From Morgan Kauffmann, San Francisco, CA, 3rd edition, 2003.
8. Hai Huang, Padmanabhan Pillai, and Kang G. Shin. From Design and implementation of power-aware virtual memory. From In *Proc. USENIX 2003 Annual Technical Conference*, pages 57–70, San Antonio, Tx, June 2003.
9. E.E. Johnson. From Graffiti on the memory wall. From *Computer Architecture News*, 23(4):7–8, September 1995.
10. Stefanos Kaxiras, Zhigang Hu, and Margaret Martonosi. From Cache decay: exploiting generational behavior to reduce cache leakage power. From In *Proc. 28th Ann. Int. Symp. on Computer architecture*, pages 240–251, G teborg, Sweden, 2001.
11. Alvin R. Lebeck, Xiaobo Fan, Heng Zeng, and Carla Ellis. From Power aware page allocation. From In *Proc. 9th Int. Conf. on Arch. Support for Programming Languages and Operating Systems (ASPLOS-9)*, pages 105–116, Cambridge, MA, November 2000.
12. P. Machanick. From The case for SRAM main memory. From *Computer Architecture News*, 24(5):23–30, December 1996.
13. P. Machanick. From Correction to RAMpage ASPLOS paper. From *Computer Architecture News*, 27(4):2–5, September 1999.
14. P. Machanick. From Scalability of the RAMpage memory hierarchy. From *South African Computer Journal*, (25):68–73, August 2000.
15. P. Machanick and Z. Patel. From L1 Cache and TLB Enhancements to the RAMpage Memory Hierarchy. From In *Proc. Eighth Asia-Pacific Computer Systems Architecture Conf.*, pages 305–319, Aizu-Wakamatsu City, Japan, September 2003.
16. P. Machanick, P. Salverda, and L. Pompe. From Hardware-software trade-offs in a Direct Rambus implementation of the RAMpage memory hierarchy. From In *Proc. 8th Int. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*, pages 105–114, San Jose, CA, October 1998.
17. Micron Technology. From 256Mb: x4, x8, x16 DDR SDRAM, December 2003. From Data Sheet, `http://download.micron.com/pdf/datasheets/dram/ddr/256Mx4x8x16DDR.pdf`.
18. NEC. From MOS integrated circuit μPD4482162, 4482182, 4482322, 4482362, December 2002. From Data Sheet No. M14522EJ3V0DS00, `http://www.necel.com/memory/pdfs/M14522EJ3V0DS00.pdf`.
19. Ashley Saulsbury, Fong Pong, and Andreas Nowatzyk. From Missing the memory wall: the case for processor/memory integration. From In *Proc. 23rd Ann. Int. Symp. on Computer architecture*, pages 90–101, 1996.
20. Hojun Shim, Yongsoo Joo, Yongseok Choi, Hyung Gyu Lee, and Naehyuck Chang. From Low-energy off-chip SDRAM memory systems for embedded applications. From *Trans. on Embedded Computing Sys.*, 2(1):98–130, 2003.
21. W.A. Wulf and S.A. McKee. From Hitting the memory wall: Implications of the obvious. From *Computer Architecture News*, 23(1):20–24, March 1995.

dDVS: An Efficient Dynamic Voltage Scaling Algorithm Based on the Differential of CPU Utilization*

Kui-Yon Mun, Dae-Woong Kim, Do-Hun Kim, and Chan-Ik Park

Department of Computer Science and Engineering/PIRL
Pohang University of Science and Technology
Pohang, Kyungbuk 790-784, Republic of Korea
{cipark}@postech.ac.kr

Abstract. Traditional dynamic voltage scaling algorithms periodically monitor CPU utilization and adapt its operating frequency to the upcoming performance requirement for CPU power management. Predicting CPU utilization is usually conducted by estimating upcoming performance requirement. In order for dynamic voltage scaling algorithms to be effective, the prediction accuracy of CPU utilization must be high. This paper proposes a power management algorithm that improves accuracy of predicting future CPU utilization using process state information. Experiments show that the proposed algorithm reduces power consumption by 11%–57% without any performance degradation.

1 Introduction

In mobile battery-powered systems, power is considered as a precious resource, and a CPU is known to consume more than 50 % of the whole system power [1]. Therefore, efficient CPU power management is required to reduce the power consumption of a system. In CPUs based on CMOS logic, the peak frequency is proportional to the supply voltage and power is proportional to the square of the supply voltage. Dynamic voltage scaling (DVS) has been implemented in most CPUs in order to control power consumption by dynamically changing its operating frequency.

Dynamic power management through DVS is classified into two approaches: an intra task approach and an inter task approach according to the location of the power management algorithm (*i.e.*, inside of a task or outside of a task). In intra-task approaches, compiler or software tool analyzes a task and determines when a CPU frequency has to be changed [2]. This approach can adjust the CPU frequency with considerable accuracy because the performance requirement of a

* The authors would like to thank the Ministry of Education of Korea for its support toward the Electrical and Computer Engineering Division at POSTECH through its BK21 program. This research was also supported in part by HY-SDR IT Research Center, and in part by grant No. R01-2003-000-10739-0 from the Basic Research Program of the Korea Science and Engineering Foundation.

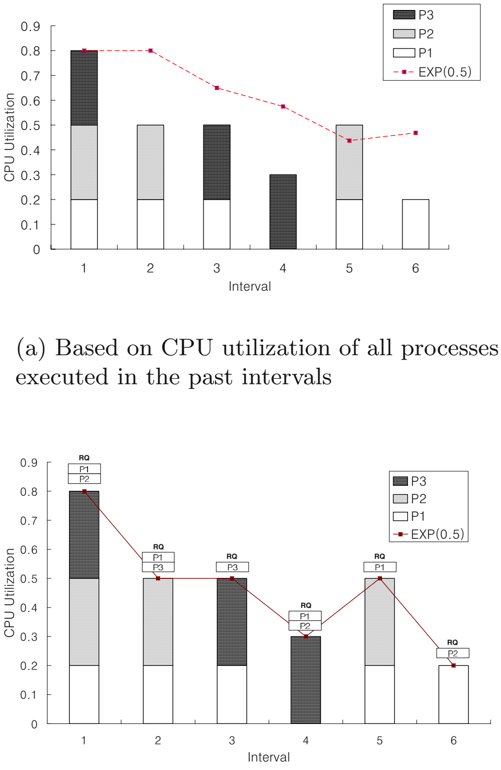
P.-C. Yew and J. Xue (Eds.): ACSAC 2004, LNCS 3189, pp. 160–169, 2004.

task is analyzed in advance. However, this approach is impractical because all applications have to be modified. Inter-task approach is divided into three approaches based on the type of information used for power management. The first approach uses the deadlines of tasks [3,4]. It achieves power reduction by considering the worst case execution time of a task and uses that information to exploit the slack time generated by the scheduler However, it requires the task deadline information, which reduces its applicability. The second approach uses the tasks' characteristics obtained from the analysis of events such as system calls and task creation/exit/switch [5]. This approach allows different power management algorithms to be applied to each task according to their characteristics. However, the overhead of event monitoring and analysis may be significant. For example, it would cost about 1% - 4% of the total CPU cycles, in order to monitor system calls and scheduler of the kernel on Transmeta Crusoe's CPU [5]. The third approach periodically monitors the CPU utilization, and uses that information to predict the expected CPU utilization [6,7]. The CPU frequency is changed adaptively to the predicted CPU utilization. Because this operation must be repeated periodically, it is called an interval-based approach. This approach provides higher applicability than other approaches while achieving simplicity. However, this approach could suffer from inefficiency due to its inaccurate prediction. We consider the interval-based approach promising because of its simplicity.

A careful investigation of existing interval-based approaches leads us to identify the reason for its inaccurate prediction. In Figure 1, we assume that there are only three processes P_1, P_2 and P_3 and their utilizations are 0.2, 0.3, and 0.3 respectively, on each run. At every interval, the CPU utilization for the next interval is computed as the exponential average (EXP) of the previous and current CPU utilizations [8]. As shown in Figure 1(a), there exists a large gap between the actual CPU utilization and the CPU utilization predicted by the EXP. We think this is mainly caused by the fact that we computed EXP over all processes regardless of their state. In Figure 1(b), after the fourth interval, P_1 and P_2 exist in the request queue (RQ). EXP predicts that P_1's utilization is 0.2 and P_2's utilization is 0.3, and computes the CPU utilization of the fifth interval as 0.5. Thus, considering the states of processes in the computation of EXP improves the prediction accuracy of the CPU utilization. Table 1 shows that there is a very low probability of a process to remain in the ready-to-run state for two consecutive intervals or more.

Table 1. Probability of a process to remain in ready-to-run state for two consecutive intervals or more: for example, the value 0.52 implies about half of the ready-to-run processes in current interval will remain in ready-to-run state until the next interval

Application type	Fraction
interactive (xpdf)	0.52
multimedia (mplayer)	0.34
I/O intensive (ubench/fsdisk)	0.93 (the fraction of cycles in which CPU is idle is 0.53)



(a) Based on CPU utilization of all processes executed in the past intervals

(b) Based on CPU utilization of processes in the ready queue (RQ)

Fig. 1. Predicting CPU utilization

This paper proposes a dynamic power management algorithm that improves the prediction accuracy of existing interval-based power management algorithms using process information. The remainder of this paper is organized as follows. Section 2 describes the proposed algorithm. Section 3 evaluates the amount of power consumption of the proposed algorithm and existing interval-based algorithms under various types of workloads. Finally, the conclusion is presented in Section 4.

2 The Proposed Algorithm

In this section, we propose an algorithm that improves the efficiency of a dynamic power management by only considering the processes that are likely going

Table 2. Symbols and their meanings

Symbol	Meaning
Q	the length of time quantum
I	the length of an interval
PUP	per-Process Utilization Predictors such as PAST, EXP and PD
f_{min}	the minimum CPU frequency
e_i/e_i^*	the actual/estimated execution time of process i in the current interval
u_i/u_i^*	actual/estimated utilization of process i in the current interval
$u_{i^-}/u_{i^-}^*$	actual/estimated utilization of process i in the previous interval
f/f^*	CPU frequency in current/next interval
P	a set of processes executed in the current interval
P^*	a set of processes to be executed with a high probability in the next interval

```
Input : $P$, $f$                    Output : $f^*$

1. Update the execution time and utilization of all processes executed in the current
   interval
     $\forall p_i \in P$
     $e_i = e_i \times \frac{f}{f_{max}}$ // for normalizing utilization by $f_{max}$ //
     $u_i = \frac{e_i}{I}$

2. Estimate CPU utilization in the next interval
     $U^* = \sum_{\forall p_i \in P^*} PUP(p_i)$,
     where $P^* = \{p_k | p_k$is a process that is going to be executed in the next interval$\}$
            and $|P^*| \leq \lceil \frac{I}{Q} \rceil$ //

3. Compute CPU frequency for next interval
     $f^* = U^* \times f_{max}$
```

Fig. 2. Description of the proposed algorithm

to be executed in the next interval. This algorithm periodically keeps track of utilizations of each process executed in previous intervals, finds the processes to be executed in the next interval, and predicts the future CPU utilization as the sum of their utilizations estimated by an existing interval-based approach. Thus, the proposed algorithm requires process information, such as the process's state, its scheduling priority and its past utilizations. Table 2 shows the description of the notations to be used in the subsequent section.

Figure 2 describes the proposed algorithm. First, it updates the utilization of all processes executed in the current interval. Utilization of process i is equal to $\frac{e_i}{I}$. The execution time of each process in this algorithm should be normalized by the maximum frequency f_{max} because it is dependent on CPU frequency when measured. For example, when the maximum frequency is 600 MHz and a process was executed at 300 MHz for 10 ms, the execution time of the process is 5 ms. Next, the proposed algorithm selects processes that are highly likely going to be executed. A process has three different states: READY to wait for run, WAIT to wait for I/O's completion or signals, and RUN to be executed on a CPU. The kernel's scheduler chooses a process with the highest scheduling priority from the set containing all processes with a ready state. Thus, the next process to be executed can be predicted by checking the priority of processes before its execution. Therefore, the proposed algorithm chooses $\lceil \frac{I}{Q} \rceil$ processes with high scheduling priority from the set of processes with READY state. The parameter $\lceil \frac{I}{Q} \rceil$ is chosen empirically. Next, it estimates the utilizations of each chosen processes. In order to predict them, it uses a per-Process Utilization Predictor (PUP). PUP predicts the utilization of a process on the next interval by using its past utilizations. Any interval-based approach can be used as PUP. Next, the proposed algorithm estimates the CPU utilization as the sum of the utilizations of each process estimated by PUP. Finally, it computes the CPU frequency in the next interval. Because the CPU utilization from the previous step is based on the minimum frequency, the CPU frequency in the next interval is equal to the multiplication of CPU utilization and the minimum frequency. The proposed algorithm applies existing interval-based approaches to each processes to be executed in the next interval instead of entire processes executed in the past intervals. Because it only considers processes with READY state, the creation/exit and state change of a process affects its prediction. Thus, in the cases where the state of a process often varies or its lifetime is short, this algorithm largely reduces the power consumption compared to existing interval-based approaches. By contrast, in the case of a process that seldom changes its state, both this algorithm and existing interval-based approaches show a similar performance.

3 Performance Evaluation

3.1 Experimental Environment

The proposed algorithm has been evaluated on a Sony VAIO-C1VJ notebook equipped with Transmeta's Crusoe CPU. The CPU provides four pairs of different frequency and voltage levels, which are (300 MHz, 1.3 V), (400 MHz, 1.35 V), (500 MHz, 1.4 V), and (600 MHz, 1.6 V). Thus, the maximum CPU frequency is 600 MHz ($f_{max} = 600$). To set the frequency and voltage, TM5600 stays in the sleep mode for 20 μsec and consumes 2 μJ–4 μJ energy. The experimental results in this paper includes the overheads such as time and power. PAST, EXP, and Proportional-Differential (PD) are used as PUPs of the proposed algorithm. PD is one of the traditional control theories and estimates the change concerning the direction of a slope [9]. PAST assumes that the future CPU utilization will be the same as the previous one. Table 3 shows the formulas of PUP when PAST, PD and EXP are used as the PUPs of the proposed algorithm.

The proposed algorithm is implemented within Linux kernel version 2.5.58 like Figure 3. The length of time quantum in Linux kernel is 10 ms ($Q = 10$). We added a new data structure, called the utilization history table, to keep track of the CPU utilization of each process. A process is allocated an entry in the table

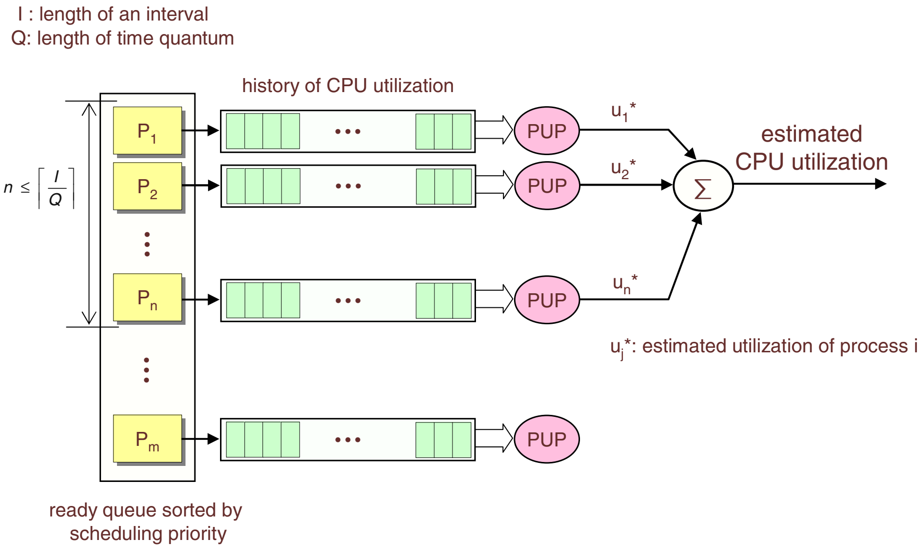

I : length of an interval
Q: length of time quantum
history of CPU utilization
$n \leq \lceil \frac{I}{Q} \rceil$
P_1
P_2
P_n
P_m
PUP
PUP
PUP
PUP
u_1^*
u_2^*
u_n^*
Σ
estimated CPU utilization
u_j^*: estimated utilization of process i
ready queue sorted by scheduling priority

Fig. 3. Implementation of the proposed algorithm within Linux kernel

Table 3. Formulas of PD, PAST and EXP being used as a per-Process Utilization Predictors

PUP	PD	$u_i^* = u_i + K_P(1 - u_i) + K_D((1 - u_i) - (1 - u_{i-})$ $(K_P = -3$ and $K_D = -3)$
	PAST	$u_i^* = u_i$
	EXP	$u_i^* = \alpha u_i + (1 - \alpha) u_{i-}^*$ $(\alpha = 0.5)$

right after its creation and releases the entry before its termination. Each entry of the table contains three fields: *valid_flag*, *curr_util* and *prev_util*. *valid_flag* indicates whether the corresponding information is valid or not, *curr_util* refers to the current utilization and *prev_util* holds the utilization of the previous periods. CPU utilization information is updated in the timer interrupt handler. The timer interrupt handler updates *curr_util* of the current process. This method has little computational overhead. More precise CPU utilization can be measured if it is updated every time processes change their states. A new kernel process is created to carry out the proposed power management. It not only periodically monitors the state of each process, but also adjusts the CPU frequency accordingly based on the predicted CPU utilization.

Table 4 shows the execution time of an application and the computing overhead when the proposed algorithm runs at different monitoring intervals. The computing overhead is calculated by taking a ratio of the execution time of the power management code to the total execution time. The interval length, denoted by *I*, is the interval at which the proposed power management code is

Table 4. The execution time of an application and the computing overhead when the proposed algorithm runs at different monitoring intervals

interval length (ms)	computing overhead (%)	execution time (sec)
10	0.67	89.9
50	0.12	90.1
100	0.06	90.0
200	0.02	124.1

executed. With an extremely short monitoring interval, the computing overhead of the algorithm increases, but the accuracy of prediction becomes high enough. For example, the 10 msec monitoring interval entails the 0.67% computing overhead shown in Table 4. By contrast, with an extremely long monitoring interval, the prediction accuracy becomes too low. For instance, Table 4 reveals that the 200 msec monitoring interval prolongs the execution time of the application by about 34 seconds, due to its inaccurate prediction. Because the number of processes on READY queue at any given moment is generally less than $\frac{I}{Q}$, so the estimated CPU utilization is prone to be less than the practical CPU utilization. The most desirable monitoring interval is observed to be 100 msec in Table 4. Note that the 100 msec monitoring interval will be used in subsequent experiments ($I = 100$).

Three types of applications have been used to verify the efficiency of the proposed algorithm. The first type is a set of I/O intensive applications. They are Ubench4.0.1/fsdisk and fstime.[1] The second type is a set of interactive applications. In order to provide identical set of inputs to the applications we logged our desired set of input using "Interactive Linux application Benchmark"[2] before carrying on with the experiments involving XPDF and LLL-1.4.[3] The third stage of the experiment was conducted using the Mplayer (Linux's MPEG player[4]), and Xmms (Linux's MP3 player[5]). In order to make a fair comparison between the power management policies, a MPEG video clip and a MP3 file are deliberately chosen so that no frame drops will occur under any policies.

3.2 Experimental Results

In order to show the prediction accuracy improvement, the proposed algorithm was compared to PAST, EXP, and PD. In the remaining sections, we abbreviated the proposed algorithm by prefixing the name of its PUP with 'P', that is, PPAST, PEXP, and PPD. The experimental results of each application are appraised by considering the power saving gains, performance impact and the

[1] `http://www.tux.org/pub/tux/benchmarks/System/unixbench`, Unix Benchmark

[2] `http://opensource.nus.edu.sg/~ctk/benchmark/bench.html`, Benchmarking of interactive linux applications

[3] `http://users.pandora.be/thomas.raes/LSS/lss.html`, Linux Lunar Lander

[4] `http://www.mplayerhq.hu/homepage/design6/info.html`, Mplayer

[5] `http://www.xmms.org`, XMMS

Table 5. Consumed power, execution time and computing overhead of power management policies during the execution of an I/O application

policy	fsdisk			fstime		
	consumed power	execution time (second)	computing overhead (μsecond)	consumed power	execution time (second)	computing overhead (μsecond)
PD	0.59	130.6	41.67	0.64	131.1	33.34
PPD	0.50	131.0	63.82	0.54	130.1	52.76
EXP	0.77	131.2	18.49	0.84	129.8	15.26
PEXP	0.33	131.2	49.01	0.41	130.0	53.65
PAST	0.52	131.3	18.55	0.54	129.9	14.81
PPAST	0.33	132.7	45.51	0.38	130.0	52.31

Table 6. Consumed power, execution time and computing overhead of power management policies during the execution of an interactive application

policy	xpdf			LLL		
	consumed power	execution time (second)	computing overhead (μsecond)	consumed power	execution time (second)	computing overhead (μsecond)
PD	0.64	115.7	31.22	0.69	23.8	70.39
PPD	0.56	119.4	46.09	0.51	24.4	96.98
EXP	0.83	111.1	13.93	0.89	25.5	25.87
PEXP	0.38	123.0	37.88	0.49	29.5	75.99
PAST	0.42	143.9	15.06	0.61	25.6	26.41
PPAST	0.62	115.1	37.24	0.39	26.9	91.23

computing overhead. When an application runs without any power management policy (NPM), we assumed that the power consumption of NPM is 1. Thus, the consumed power is normalized by that of NPM. The computing overhead is calculated by taking the average of execution times of the power management code over all intervals.

As shown in Table 5, the performance of I/O applications can be assessed by their execution time. PPD, PEXP, and PPAST achieved a 15%–17%, 51%–57% and 30%–37% power reduction over PD, EXP, and PAST respectively. An I/O intensive process repeats the WAIT-READY-RUN cycle to handle I/O requests. By considering the current states of the processes, we can figure out when an I/O intensive process is going to take up a portion of the CPU time. This results in reduced power consumption of I/O applications by 15%–57% without delay in execution time.

Table 6 shows the experimental results when an interactive application, xpdf or LLL, generates workload. In general, the proposed algorithm reduces power consumption by 12.5%–54%. However, in the case of xpdf, PPAST consumes more power than PAST. Note that the PAST increases the execution time of the xpdf compared to other policies by 32.8 seconds in the worst case. This is

Table 7. Consumed power, execution time and computing overhead of power management policies during the execution of an multimedia application

policy	mplayer		xmms	
	consumed power	computing overhead (μsecond)	consumed power	computing overhead (μsecond)
PD	0.56	57.28	0.43	25.33
PPD	0.50	83.57	0.36	49.70
EXP	0.81	25.61	0.65	34.60
PEXP	0.42	69.53	0.35	46.83
PAST	0.64	25.68	0.47	38.70
PPAST	0.41	71.80	0.37	50.92

a severe performance degradation. It seems that PAST predicts a lower CPU utilization than the actual CPU utilization.

Table 7 shows the experimental results when the multimedia file chosen is played with any frame drops. The proposed algorithm diminished the power consumption by 11%–48% over PAST, EXP, and PD. However, the power reduction rate of the multimedia application is less than that of the other applications. Because the multimedia application process stays in READY and RUN state after its creation, a slight gap in the power reduction rate occurs.

As shown in the experimental results of Table 5–6, the proposed algorithm achieves a large power reductions over existing interval-based approaches, such as PAST, EXP and PD. Although the computing overhead increases to 3.5 times in the worst case, each application runs without any delay in its execution time and the power consumption also decreases as well.

4 Conclusion and Future Work

This paper proposed an efficient power management algorithm in order to improve prediction accuracy using processes' state information. The experiments with the proposed algorithm revealed that the power consumption of the proposed algorithm is reduced by 11%–57% when compared to the existing interval-based algorithms such as PD, EXP and PAST. However, the efficiency of the proposed algorithm is depended on the length of the monitoring interval. Choosing the optimal interval length for each type of applications still remains as our future work.

References

1. COMPAQ: Compaq presario based on amd mobile athlon 4 (2001)
2. Azevedo, A., Issenin, I.: Profile-based dynamic voltage scheduling using program checkpoints. In: Proceedings of design automation and test in Europe. (2002)

3. Krishna, C.M., Lee, Y.H.: Voltage-clock-scaling adaptive scheduling techniques for low power and real-time systems. In: Proceedings of the sixth IEEE real time technology and applications symposium. (2000)
4. Okuma, T., Ishihara, T., H.Yasuura: Real-time task scheduling for a variable voltage processor. In: Proceedings of the international symposium on system synthesis. (1999)
5. Flautner, K., Mudge, T.: Vertigo: automatic performance-setting for linux. In: Proceedings of operating systems design and implementation. (2002)
6. Govil, K., Chan, E., Wasserman, H.: Comparing algorithms for dynamic speed-setting of a low-power cpu. In: Proceedings of the first international conference on mobile computing and networking. (1995)
7. Pering, T., Burd, T., Brodersen, R.: The simulation and evaluation of dynamic voltage scaling algorithms. In: Proceedings of international symposium on electronics of lower power and design. (1998) 76–81
8. Lu, Y.H., Benini, L.: Power-aware operating systems for interactive systems. IEEE Trans. VLSI **10** (2002) 119–134
9. Kuo, B., Golnaraghi, F.: Automatic control systems. eighth edn. John Wiley & Sons, Inc. (2003)

High Performance Microprocessor Design Methods Exploiting Information Locality and Data Redundancy for Lower Area Cost and Power Consumption

Byung-Soo Choi[1], Jeong-A Lee[2], and Dong-Soo Har[3]

[1] Ultrafast Fiber-Optic Networks Research Center
K-JIST(Kwangju Institute of Science and Technology)
1 Oryong-dong Puk-gu Gwangju, 500-712, Republic of Korea
bschoi@kjist.ac.kr

[2] Department of Computer Engineering
Chosun University
375 Susuk-dong Dong-gu Gwangju, 501-759, Republic of Korea
jalee@chosun.ac.kr

[3] Department of Information and Communications
K-JIST(Kwangju Institute of Science and Technology)
1 Oryong-dong Puk-gu Gwangju, 500-712, Republic of Korea
hardon@kjist.ac.kr

Abstract. Value predictor predicting result of instruction before real execution to exceed the data flow limit, redundant operation table removing redundant computation dynamically, and asynchronous bus avoiding clock synchronization problem have been proposed as high performance microprocessor design methods. However, these methods increase area cost and power consumption problems because of the larger table for value predictor and redundant operation table, and the higher switching activity in asynchronous bus. To resolve the problems of data tables for value predictor and redundant operation table, we have investigated partial tag and narrow-width operand methods, which have been recently proposed separately and present an efficient update method for value predictor and a table organization method for redundant operation table, respectively. To reduce excessive switching activity of asynchronous bus, we also propose a bus encoding method using frequent value cache, which reduces the same data transmissions. The proposed three methods – an efficient update method for value predictor, a table organization method for redundant operation table, and a frequent value cache for asynchronous bus – exploit information locality such as instruction and data locality as well as data redundancy. Analysis with a conventional microprocessor model show that the proposed three methods reduce total area cost and power consumption by about 18.2% and 26.5%, respectively, with negligible performance variance.

P.-C. Yew and J. Xue (Eds.): ACSAC 2004, LNCS 3189, pp. 170–184, 2004.

1 Introduction

Until a few years ago, performance improvement has been a key research issue in microprocessor design. Recently, however, the area cost and the power consumption of a microprocessor have been increased drastically as the number of transistors keeps increasing. As a result, research interest has been shifted to performance improvement while maintaining the efficiency of area cost and power consumption. In this paper, several design methods have been investigated for a high performance microprocessor with an emphasis on achieving efficient area cost and power consumption.

Among many design techniques for a high performance microprocessor, three methods are investigated such as value predictor, redundant operation table, and asynchronous dual-rail bus in this research. The value predictor predicts a result of an instruction before the instruction is actually executed. Hence dependent instructions can be executed at the same time when the instruction is executed. On the other hand, the redundant operation table stores recently executed instructions in a table and checks whether the current executable instruction is already stored in the table. In other words, the redundant operation table can skip the real execution of an instruction by a simple lookup procedure with the table, subsequently shortening the execution time of the instruction. Another alternative design technique, the asynchronous dual-rail bus is a reliable bus scheme for a complex system such as a futuristic high performance microprocessor. The asynchronous dual-rail bus can transmit data in a reliable fashion by making use of the dual-rail encoding, which combines the data and the control signals.

Analyzing the aforementioned three design methods from the area cost and power consumption points of view, several attempts are made especially to find some locality and redundancy of data used in each design method. Several information localities and data redundancies were found, which causes extra area cost and power consumption. More specifically, the value predictor and the redundant operation table store the same or a little different instructions (instruction locality), small operand values (operand data locality), and small result values (result data locality), whereas the asynchronous dual-rail bus transmits the same data items repeatedly (communication data locality). From what we observed about these localities, a conclusion was reached that each design method can be further enhanced for lower area cost and lower power consumption by exploiting such localities to reduce redundancy.

In this paper, we propose three enhanced methods as follows. First, for value predictors, we propose a method to combine the two previously proposed area cost reduction methods such as partial-tag and narrow-width methods. Second, we designed a partial resolution method to reduce the area cost of the tag fields in the redundant operation table. Third, we applied the previously proposed frequent value cache method into an asynchronous dual-rail bus to minimize the communication data redundancy.

As the last step, we investigated total area cost and power consumption reduction effects in a conventional microprocessor model. By using the proposed methods, the total area cost and power consumption in a microprocessor model would be reduced by about 18.2% and 26.5%, respectively.

This paper is organized as follows. Section 2 describes related work as three high performance design methods, information locality, and data redundancy. The proposed area and power reduction methods for value predictor and redundant operation table are described in Section 3 and 4, respectively. Also a designed power reduction method for asynchronous dual-rail bus is explained in Section 5. Meanwhile, total area cost and power consumption reduction effects in a microprocessor model are analyzed in Section 6. Section 7 concludes this research.

2 Related Work

2.1 High Performance Design Methods

Value Predictor: Value predictors have been proposed to overcome the data dependency problems in the instruction-level parallelism by predicting a result value of an instruction before its actual execution [1], [2].
Redundant Operation Table: When the instruction-level parallelism increases, there are many side effects. One of the side effects is the increased number of redundant executions because of speculative executions due to branch predictor or value predictor. Unfortunately, speculative or redundant operations limit the performance improvement and increase the power consumption as well [3]. To overcome such negative effects, many optimization methods have been proposed [4], [5]. One typical solution is eliminating redundant operations, where redundant executions of complex operations are replaced by simple table lookup operations [6].
Asynchronous Dual-Rail Bus: Because of the steady increase of the number of components in a chip, SOC design methods have been studied intensively and will be used for a futuristic high performance microprocessor. To succeed in the market, the time-to-market and the reliability of a SOC are very important. To help the design efforts for a short design time and reliability of SOCs, asynchronous design methods [7] have been studied recently. For a reliable asynchronous bus structure in SOC designs, the dual-rail data encoding method [8] has been intensively investigated.

2.2 Information Locality and Data Redundancy

Information Locality: Information localities in a microprocessor are defined, which are related to instructions, operands of instructions, results of instructions, and communication data over bus. First, *instruction locality* is defined as a small number of instructions is repeatedly or frequently executed, and usually the instructions are located closely to each other in a given time interval. Second, *operand locality* is defined as the data value of the operand is small in most instructions and can be represented with small number of bits. Third, *result locality* is defined as the results of most instructions are small which can be represented with small number of bits. Last, *communication data locality* is defined as a bus transmits the same or very similar data repeatedly or frequently in a given time interval.

Data Redundancy: Considering the above information localities, we can infer that there are data redundancy in instructions, operands of instructions, results of instructions, and communication data over bus, respectively. First, *data redundancy of instructions* is occurred when the instruction addresses in a given time interval are not so different, which can be inferred that the higher bits of instruction address are the same. Hence the higher bits of addresses of executed instructions in a given time interval are redundant. Second, *data redundancy of operands* is occurred when most operands of executed instructions in a given time interval require a small number of bits, and hence the higher bits of operands are considered as redundant bits. Third, *data redundancy of results* is occurred when the most results of executed instructions in a given time interval require a small number of bits, and hence the higher bits of results are redundant. Last, *data redundancy of communication data* is occurred when the most communication data in a given time interval are the same or similar, and hence most communications are redundant.

3 Value Predictor

3.1 Table Structure

In this research, we explain only the stride predictor for the simplicity. The stride predictor assumes that consecutive result values of an instruction have the same stride value [1]. Usually, a value predictor exploits a large data table to store required information and is referenced by the instruction address.

3.2 Combining Partial Tag and Narrow-Width Operand Method

Two Methods to Reduce Area and Power of Value Predictor: To reduce the area cost and power consumption of value predictor, two methods have been already proposed as follows.

Partial Tag Method: Instruction or data caches are usually based on a correct association between an instruction address and an indexed entry because the lookup data must be the same value as the previously stored value. In the value predictor, however, a lookup data is a prediction value so that it does not always require the correct association between a lookup address and an indexed data. Based on such a loose association between a lookup address and an indexed data, a value predictor does not necessarily use a full-tag, but can use a partial-tag, which reduces the area cost of the tag part [9]. Briefly, the full-tag method takes an address as a tag except for index bits, but the partial-tag method only uses some part of a full-tag.

Narrow-Width Operand Method: Analysis of the result values of a program shows that only a few result values require a full precision value supported by processor

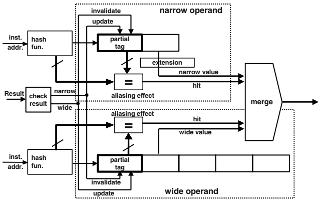


Fig. 1. Combining Method of Partial-Tag and Narrow-Width Methods

registers. Taking into account such locality, the narrow-width operand method classifies result values into two types as the narrow-width and the wide-width result values according to the required number of bits [10]. For the purpose of area cost reduction in data tables, the narrow-width operand method utilizes both the narrow-width and wide-width tables for storing the narrow-width and wide-width result values, respectively. If a result value of an instruction requires fewer bits than the predetermined number of bits, prediction information of the instruction is stored in the narrow-width table. Otherwise, prediction information is stored in the wide-width table. Because the narrow-width table stores fewer bits for each result value, it reduces the overall area cost of a data table.

Combining Partial Tag and Narrow-Width Operand Methods: To date, two area cost reduction methods for value predictors have been proposed independently. In the present research, a combining method with an efficient table-update method is proposed to minimize the performance degradation. A simple method combining these two methods is conceivable. However, such a simple superposition method decreases the performance improvement ratio because two prediction values are generated from the two tables.

We propose a new table-update method as shown in Figure 1. When the result of an instruction is classified into a narrow-width result(wide-width result), the instruction is stored in the narrow-width table(wide-width table). At the same time, the wide-width table(narrow-width table) invalidates an indexed entry if the entry contains the same partial-tag with the instruction. In short, depending on the classification result of an instruction, only one of the two tables stores the instruction, and the other table must invalidate a corresponding entry if the tag is the same with the referenced address.

3.3 Analysis

To measure the effect of the proposed area reduction method, the die size and the power consumption of value predictor are measured by using CACTI 3.2 [11]. We also investigated IPC value when the proposed method is used with a SimpleScalar [12] model and SPEC 95 [13] benchmark programs. However, as we expected, the IPC value changes very little about 1%. Hence we skip the explanation of IPC variation when the proposed method is used.

Area Cost: Table 1 describes area cost reduction ratios over the conventional stride predictor. The reduction of area cost is higher with the narrow-width

Table 1. Area Cost and Power Consumption Reduction Ratios with Different Area Reduction Methods for Stride Predictor

Area Reduction Methods	Area Cost				Power Consumption			
	Number of Entries (K)				Number of Entries (K)			
	32	16	8	4	32	16	8	4
Narrow-Width	57%	55%	56%	47%	16%	17%	27%	24%
Partial-Tag	20%	19%	20%	20%	39%	31%	46%	49%
Proposed Combining Method	***72%***	***74%***	***72%***	***68%***	***58%***	***51%***	***63%***	***68%***

method than with the partial-tag method. The reason is as follows. The reduction ratio depends on the portion of the area cost reduced by the partial-tag and the narrow-width methods. The partial-tag method can decrease the area cost of the tag part only; however, the narrow-width method can decrease the area cost of all result values. Meanwhile, the proposed combining method decreases the area cost more than other area cost reduction methods. The proposed combining method decreases the area cost by about 71% for the stride predictor.

Power Consumption: Table 1 also describes power consumption reduction ratios over the conventional stride predictor. The reduction of power consumption is higher with the partial-tag method than with the narrow-width method. The reason is as follows. The power consumption of the tag part is larger than that of data part since each tag comparison requires more power consumption. Meanwhile, the proposed combining method decreases the power consumption more than other area cost reduction methods. The proposed combining method reduces the power consumption by about 61% for the stride predictor.

4 Redundant Operation Table

4.1 Table Structure

In a redundant operation table, operands are partitioned into two parts: an index and a tag parts. Meanwhile, all operations are classified into integer or floating-point operations. Hence redundant operation tables have different structures depending upon the operation type.

4.2 Narrow-Wide-Width Table

A preliminary analysis of operands for integer and floating-point operations in a SimpleScalar [12] microprocessor with SPEC [13] benchmarks reveals that most operands can be represented with a small number of bits. A partial resolution method is proposed to exploit this characteristics. The partial resolution method eliminates the area cost to store redundant bits for consecutive 0s in the higher bits for integer operands and the lower bits for floating-point operands in the conventional wide-width redundant operation table. A wide-narrow-width redundant operation table utilizing the partial resolution method is designed as shown in Figure 2. The wide-narrow-width redundant operation table dynamically classifies operations into wide-width and narrow-width operations depending on the

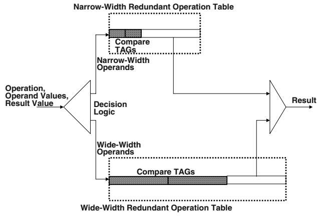

Narrow-Width Redundant Operation Table
Compare TAGs
Narrow-Width Operands
Operation, Operand Values, Result Value
Decision Logic
Result
Wide-Width Operands
Compare TAGs
Wide-Width Redundant Operation Table

Fig. 2. Wide-Narrow-Width Redundant Operation Table

operand bit width. When the operation requires narrow-width operands, the instruction is stored in the narrow-width redundant operation table. Otherwise, the instruction is stored in the wide-width redundant operation table. Note that the concept of the partial resolution method is similar to the partial-tag method [9], which is proposed to apply for value predictors. The partial-tag method for value predictors stores imprecise tag information, but the partial resolution method for redundant operation table should store precise tag information. Hence, the partial-tag method for value predictor cannot be directly used for the redundant operation table.

4.3 Analysis

Note that we also investigated IPC value when the proposed method is used with a SimpleScalar [12] model and SPEC 95 [13] benchmark programs. However, since the IPC variance is very little, we skip the explanation of IPC variation when the proposed method is used.

Area Cost: The area cost of the conventional wide-width redundant operation table can be calculated easily. Meanwhile, the wide-narrow-width redundant operation table consists of two subsidiary predictors, hence the area cost of it is calculated by the summation of each area cost for narrow-width and wide-width tables. Based on the above considerations and methods, the relative area cost is measured as shown in Table 2. Note that the models containing above 512 entries are measured, since the redundant operation table usually requires many entries. As the table explains, the proposed partial resolution method reduces the area cost by 20%, for FP 2048-entry, at the maximum.

Power Consumption: Since the conventional wide-width redundant operation table is referred for all lookups, it can be easily calculated the dynamic power consumption of the wide-width table. On the other hand, since each subsidiary table in the wide-narrow-width redundant operation table is referred with different lookup ratios, the lookup ratio of each table should be considered. Hence the total dynamic power consumption of the proposed wide-narrow-width redundant operation table is calculated by the summation of each power consumption of narrow-width and wide-width tables considering each lookup ratios. Based on the above considerations and methods, the relative dynamic power consumption reduction ratio is measured as shown in Table 2. As the table explains, the

Table 2. Relative Area Cost and Power Consumption Reduction Ratio

Reduction Ratio over Wide-Width Table		Number of Entries	
		2048	512
Area Cost	INT	7%	9%
	FP	20%	10%
Power Consumption	INT	34%	24%
	FP	30%	31%

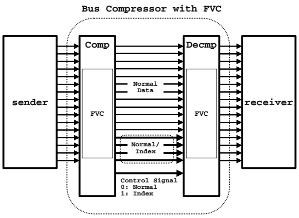


Fig. 3. Frequent Value Cache augmented Bus Scheme

proposed partial resolution method reduces the dynamic power consumption by about 34%, for INT 2048-entry, at the maximum.

5 Asynchronous Dual-Rail Bus

5.1 Frequent Value Cache

One-of-four data encoding method reduces the power consumption of the dual-rail encoding method by decreasing switching activities [14]. Meanwhile, the data pattern analysis illustrates that many data items are repeatedly transmitted in accordance with the result in [15]. Hence we can conclude that the conventional dual-rail and the previously proposed one-of-four data encoding methods waste the power when the data bus transmits the same data items repeatedly.

To reduce such waste of power, we proposed a different method, which utilizes a buffer to exploit the feature of repeatedly transmitted data item. The proposed buffer stores data items and sends an index for a data item when the data item to be sent is already stored in the buffer. Since the index requires fewer number of bits than the data itself, the wasted bandwidth or the switching activity can be decreased, resulting in low power consumption.

Figure 3 describes a frequent value cache(FVC) very briefly that stores data items of each communication. The normal *sender* and *receiver* deliver a data item with a normal fashion, while the *Comp* and *Decmp* deliver a data item by a data itself or an index of *FVC* depending on the hit of *FVC*. When a data itself is transferred, all bus lines are used; however, when an index of the data item is transferred, only the index lines are used. Thus, the index lines are used for both an index and a data item. To distinguish whether a transmitted information represents an index or a normal data item, a *control* signal is used.

5.2 Analysis

Three measures as hit ratio, switching activity reduction ratio, and power consumption reduction ratio are investigated. The hit ratio is the most important one since it decides the switching activity reduction ratio that finally determines the power consumption reduction ratio. To analyze, we investigated a memory bus in SimpleScalar model [12] and SPEC95 benchmark [13] programs.

Hit Ratio: We found the following conclusions through investigating data patterns over the above memory bus. First, even only one entry of FVC can detect 40% of the repeatedly transmitted data items. Second, over 256 entries can represent most data items.

Switching Activity: From the high hit ratio of the FVC, it is required to know how much switching activity can be reduced. In the research, only the change of signal levels between consecutive data items are measured to calculate the switching activity ratio of a bus. The normal dual-rail bus utilizes all 32-bit logical bits and each signal line causes two switchings, hence the switching activity is 32 × 2. Meanwhile, FVC delivers an index for a hit case and a normal data item for a miss case. In addition, the control signal changes for every communications, hence it changes two times for each communication. Therefore, the switching activity when FVC is used is calculated by Equation 1.

$$P_{hit} \times \{1 + \log(\#entry)\} \times 2 + (1 - P_{hit}) \times (1 + 32) \times 2 \tag{1}$$

Based on the above analysis, the switching activity reduction ratio of FVC over the normal dual-rail bus model is calculated by Equation 2.

$$\frac{P_{hit} \times \{1+\log(\#entry)\} \times 2 + (1-P_{hit}) \times (1+32) \times 2}{32 \times 2} \tag{2}$$

Analysis result illustrates that FVC reduces the switching activity of the conventional model by 75% at maximum. However, the switching activity reduction ratio is decreased after the maximum point because of the increased number of index bits.

Power Consumption: The total power consumption should include the power consumption of the FVC tables although the power consumption ratio of the table would be below 5% as explained in [16]. In addition, the power consumption of the bus itself should be considered as well. To measure the power consumption of FVC table and bus lines, it is assumed that 0.25 micron technology is used, and the length of the bus line is 10 mm, which follows the 2001 ITRS [17]. Power consumptions of the normal model and the FVC model are as follows:

Normal Model: The power consumption is only caused by the dual-rail bus for logical 32-bit bus lines. Based on the 0.25 micron technology, we assume that 10 mm bus lines consume about 0.4 nJ by using power measure tools.

FVC Model: The power consumption is caused by two parts as the FVC table and bus lines. To measure the power consumption of the FVC table, CACTI tool [11] is used. Since all entries should be checked at the same time, it is assumed that the table is a fully-associative content address memory. The power consumption of FVC model can be formulated as Equation 3. Specifically, the

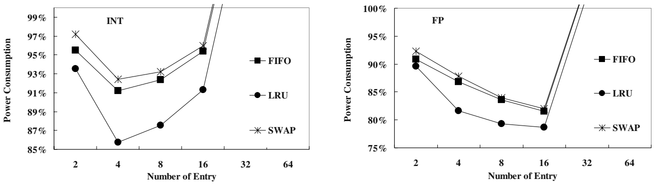


Fig. 4. Power Consumption Variation

power consumption of the FVC table is multiplied by two because FVC model requires two FVC tables for a sender and a receiver. Meanwhile, when the FVC miss, each FVC table must be updated and it consumes more power. To include this power consumption to update FVC, we include the *Miss_Ratio* in the equation.

$$\begin{aligned}&Table\_Power \times 2 \times (1 + Miss\_Ratio)\\&+Bus\_Power \times Switching\_Activity\_Reduction\_Ratio\end{aligned} \tag{3}$$

Finally, it can be derived a power consumption reduction ratio of the FVC model over the normal model as shown in Equation 4.

$$\frac{Table\_Power \times 2 \times (1+Miss\_Ratio)+(0.4nJ)\times Switching\_Activity\_Reduction\_Ratio}{0.4nJ} \tag{4}$$

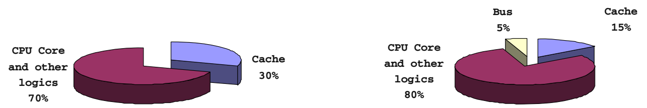
Figure 4 shows the power consumption reduction ratio when the FVC model is used. From the figure, it can be concluded that FVC reduces the total power consumption by about 14% and 22% at maximum for integer and floating-point benchmarks, respectively.

6 Analysis in a Microprocessor

Until previous sections, it have been analyzed independently the area cost and/or power consumption reduction ratios of the proposed methods for value predictor, redundant operation table, and asynchronous dual-rail bus. Meanwhile, because our main goal is to reduce the total area cost and power consumption of a high performance microprocessor, it is needed to know how much area cost and power consumption can be reduced when the proposed methods are used for each design method.

6.1 Area Cost and Power Consumption Breakdowns

Because no processor has been implemented with the value predictor, redundant operation table, and asynchronous dual-rail bus at the same time, it is required to model a futuristic microprocessor to investigate the portions of area cost and power consumption of each design method.



(a) Area Cost

(b) Power Consumption

Fig. 5. Area Cost and Power Consumption Breakdowns of Alpha 21264 Model

Conventional Model: The Alpha 21264 microprocessor [18], [19] is selected to find the breakdown of die size and power consumption of major blocks such as cache and core parts. Because value predictor and redundant operation table have similar structure with the cache, it can be assumed that the area cost and power consumption of tables for value predictor and redundant operation table are calculated by the relative area cost and power consumption over the cache.
Area Cost and Power Consumption Breakdown: Alpha 21264 utilizes 128Kbyte Instruction and Data caches, which require about 30% of total area cost [18] and consumes about 15% of total power consumption [19]. Figure 5(a) and 5(b) show the breakdowns of area cost and power consumption of the Alpha 21264 model, respectively.
New Model: The new Alpha 21264 model consists of the old Alpha 21264 and other three design methods. Because of such modification of the old Alpha 21264 model, the area cost and power consumption breakdowns will be changed.
Area Cost Breakdown: The area cost of caches is about 30% and the others about 70% in the old Alpha 21264 processor. However, the value predictor and redundant operation tables add more area cost as 164Kbyte and 144Kbyte, respectively. In the old Alpha 21264 processor, 128Kbyte cache uses about 30% of total die size, hence it can be inferred that the value predictor increases the total area cost by about 38.4%, which is calculated by 30%*164/128. Also, the redundant operation table adds about 33.8% of total area cost, which is calculated by 30%*144/128. Finally, the total area cost is increased by about 72.2%, which is calculated by the summation of the extra area cost of value predictor and redundant operation table. From this total area cost increase, it should be rearranged the portion of area cost of each component as 17.4% for cache, 22.3% for value predictor, 19.6% for redundant operation table, and 40.7% for others as shown in Table 3. As shown in the table, it can be known that the portions of area cost for value predictor and redundant operation table are large, about 42%.
Power Consumption Breakdown: On the other hand, the portions of additional power consumption of value predictor and redundant operation table can be calculated by the relative power consumption over cache. It is inferred that the stride type value predictor consumes five times as much energy as cache from [4]. Hence, the value predictor consumes more energy by about 96.1%, which is calculated by 15%*(164/128)*5. Meanwhile, the redundant operation table also consumes more energy by about 16.9%, which is calculated by 15%*(144/128). From this increased total power consumption, it should be rearranged the portions of power consumption of each block as 7.1% for cache, 2.3% for bus, 45.1% for value

Table 3. Area Cost Breakdown, Reduction Ratios, Relative Reduction in the new Alpha 21264

Parts	Portion	Reduction Ratio	Relative Reduction
Cache	17.4%		
CPU Core	40.7%		
Value Predictor	**22.3%**	**64%**	**14.3%**
Redundant Operation Table	**19.6%**	**20%**	**3.9%**
Total	100%		**18.2%**

Table 4. Power Consumption Breakdown, Reduction Ratios, Relative Reduction in the new Alpha 21264

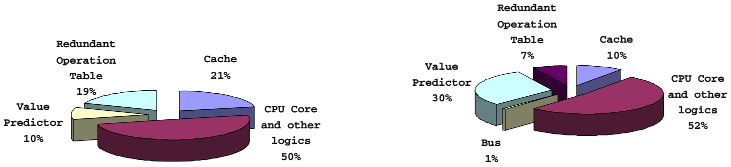
Parts	Portion	Reduction Ratio	Relative Reduction
Cache	7.1%		
CPU Core	37.6%		
Bus	**2.3%**	**14%**	**0.3%**
Value Predictor	**45.1%**	**52%**	**23.5%**
Redundant Operation Table	**7.9%**	**34%**	**2.7%**
Total	100%		**26.5%**

predictor, 7.9% for redundant operation table, and 37.6% for CPU Core as shown in Table 4. The extra power consumption caused by value predictor, redundant operation table, and asynchronous dual-rail bus is very large, about 55%.

6.2 Reduction of Total Area Cost and Power Consumption

Reduction of Total Area Cost and Power Consumption:
Area Cost Reduction: When the proposed area cost reduction methods for value predictor and redundant operation table are used, the total area cost can be reduced by about 18.2%, which is calculated by the summation of reduction ratios of area cost for value predictor (22.3% * 64% = 14.3%) and redundant operation table (19.6% * 20% = 3.9%), as shown in Table 3.
Power Consumption Reduction: Meanwhile, the proposed power consumption reduction methods can decrease the power consumption of each design method by about 52% for value predictor, 34% for redundant operation table, and 14% for asynchronous dual-rail bus, which are shown in Table 4. Therefore, the proposed area cost and power consumption reduction methods reduce the power consumption by about 23.45%(=45.1%*52%), 2.7%(=7.9%*34%), and 0.3%(=2.3%*14%), respectively, and finally the total power consumption by about 26.5% as shown in Table 4.

Area Cost and Power Consumption Breakdowns:
Area Cost Breakdown: The portions of area cost of value predictor and redundant operation table are changed as shown in Figure 6(a). As shown in the figure, the total portion of area cost for value predictor and redundant operation tables is reduced from 42% to 29%.



(a) Area Cost

(b) Power Consumption

Fig. 6. Area Cost and Power Consumption Breakdowns of Area Cost Reduced Alpha 21264 Model

Power Consumption Breakdown: The portions of power consumption of value predictor, redundant operation table, and asynchronous dual-rail bus are changed as shown in Figure 6(b). As shown in the figure, it can be known that the total portion of power consumption of value predictor, redundant operation table, and asynchronous dual-rail bus is reduced from 55% to 38%.

7 Conclusion

Throughout this research, we have pointed out that the low area and power design methods should be proposed for design techniques for a high performance microprocessor. Among many techniques, three high performance design techniques have been investigated.

Analysis of information locality and related data redundancy illustrates that the area and power are wasted by the data redundancy in each high performance design method. Therefore, the information locality was exploited and tried to minimize data redundancy in each method. Finally, three different approaches have been proposed for each method respectively.

First, to reduce the waste of area cost and power consumption in a value predictor, which is caused by data redundancy in tag and data part, we proposed a combining method of previously proposed partial tag and narrow-width method with an efficient table-update method. Structural and dynamic analysis show that the proposed method reduces the area cost by about 71% and the power consumption by about 61% over the conventional value predictor. Second, for the redundant operation table, we designed a partial tag method. Although the redundant operation table wastes area and power in both tag and data parts, a redundancy minimization method only for tag part has been discussed. The proposed method reduces the area cost by about 20% and the power consumption by about 34% over the ordinal redundant operation table structure. Third, to reduce the waste of power consumption of asynchronous dual-rail bus, we utilize the frequent value cache with several circuits. Analysis results show that the proposed method decreases the power consumption of a bus in a microprocessor by about 14% for integer and 22% for floating-point data communications over a memory bus in a microprocessor.

As well, we examined how much total area cost and power consumption can be reduced when the proposed area cost reduction methods are used for each design method. This analysis confirmed that the total area cost and power consumption would be reduced by about 18.2% and 26.5%, respectively.

Acknowledgments. This work was supported in part by the Korea Science and Engineering Foundation (KOSEF) through the Ultra-Fast Fiber-Optic Networks Research Center at Kwangju Institute of Science and Technology.

References

1. Mikko H. Lipasti and John P. Shen: Exceeding the Dataflow Limit via Value Prediction, Proc. of 29th Intl. Symp. on MICRO, (1996) 226-237
2. Sang-Jeong Lee, Yuan Wang, and Pen-Chung Yew: Decoupled Value Prediction on Trace Processors, Proc. of 6th IEEE Intl. Symp. on HPCA (2000) 231-240
3. Rafael Moreno, Luis Pinuel, Silvia del Pino, and Francisco Tirado: A Power Perspective of Value Speculation for Superscalar Microprocessors, Proc. of ICCD, (200) 147-154
4. Ravi Bhargava and Lizy K. John: Latency and Energy Aware Value Prediction for High-Frequency Processors, Proc. of the 16th ICS (2002) 45-56
5. Ravi Bhargava and Lizy K. John: Performance and Energy Impact of Instruction-Level Value Predictor Filtering, Proc. of the First Workshop of Value-Prediction (2003)
6. Daniel Citron and Dror G. Feitelson: Hardware Memoization of Mathematical and Trigonometric Functions, Technical Report-2000-5, Hebrew University of Jerusalem (2000)
7. Scott Hauck: Asynchronous Design Methodologies: An Overview, Proc. of the IEEE (1995) Vol.83, No.1, 69-93
8. Tom Verhoeff: Delay-Insensitive Codes: An Overview, Distributed Computing (1988) Vol.3, 1-8
9. Toshinori Sato and Itsujiro Arita: Partial Resolution in Data Value Predictors, Proc. of ICPP (2000) 69-76
10. Toshinori Sato and Itsujiro Arita: Table Size Reduction for Data Value Predictors by exploiting Narrow Width Values, Proc. of ICS (2000) 196-205
11. Premkishore Shivakumar and Norman P. Jouppi: CACTI 3.0: An Integrated Cache Timing, Power, and Area Model, WRL Research Report 2001/2, COMPAQ Western Research Laboratory (2001)
12. Doug Burger and Todd M. Austin: The SimpleScalar Tool Set, Version 2.0, Technical Report, CS-TR-97-1342, University of Wisconsin (1997)
13. SPEC CPU Benchmarks: Standard Performance Evaluation Cooperation http://www.specbench.org/osg/cpu95
14. John Bainbridge and Steve B. Furber: Delay Insensitive System-on-Chip Interconnect using 1-of-4 Data Encoding, Proc. of ASYNC. (2001) 118-126
15. Benjamin Bishop and Anil Bahuman: A Low-Energy Adaptive Bus Coding Scheme, Proc. of the IEEE Workshop of VLSI (2001) 118-122
16. Tiehan Lv, Jorg Henkel, Haris Lekatsas, and Wayne Wolf: An Adaptive Dictionary Encoding Scheme for SOC Data Buses, Proc. of DATE (2002) 1059-1064

17. The Semiconductor Industry Association: The International Technology Roadmap for Semiconductor (2001)
18. Srilatha Manne, Artur Klauser, and Dirk Grunwald: Pipeline Gating: Speculation Control for Energy Reduction, Proc. of ISCA (1998) 122-131
19. Michael K. Gowan, Larry L. Biro, and Daniel B. Jackson: Power Considerations in the Design of the Alpha 21264 Microprocessor, Proc. of DAC (1998) 726-731

Dynamic Reallocation of Functional Units in Superscalar Processors

Marc Epalza[1], Paolo Ienne[2], and Daniel Mlynek[1]

[1] Laboratoire de Traitement des Signaux 3,
[2] Laboratoire d'Architecture des Processeurs,
Swiss Institute of Technology Lausanne (EPFL), 1015 Ecublens, Switzerland
{marc.epalza, paolo.ienne, daniel.mlynek}@epfl.ch

Abstract. In the context of general-purpose processing, an increasing number of diverse functional units are added to cover a wide spectrum of applications. However, it is still possible to design custom logic adapted to a particular application that will perform far better than a processor. In an attempt to give it some adaptability, adding some reconfigurability can help improve performance. We propose to extend the possibilities of complex multifunction units by dynamically reallocating existing complex functional units as multiple simpler units. The fact that more than one simple unit is involved in the ”reconfiguration” process implies that the decision is more global and needs to be taken for a longer period of time. We show that in typical superscalar architectures, there are no major impediments to implementing such a decision scheme, and that on a specific reallocation opportunity we can achieve speedups of up to 56% over a mainstream superscalar processor and practically no losses.

1 Introduction

In general purpose processors, the quest for ever higher performance leads to many trade-offs, since one aims to achieve the best average performance on a variety of tasks essentially unknown to the designer. Many methods to extract even more parallelism, such as speculative execution or *Very Long Instruction Word* (VLIW) compiler technologies are complex and achieve diminishing returns, since the resources available to the processor are fixed. Attempts to make the processor adaptable to the program it is currently executing, through the use of reconfigurable logic, have provided mixed results. We propose to introduce some adaptability without using slow reconfigurable logic. To this end, we focus on the large multi-function units present in a superscalar processor. As an example, we expose a modification of a superscalar processor's *floating point functional units* (FPU) to allow some adaptation to the current workload.

Section 2 will lay out the constraints of the field and existing methods to achieve high performance. Next, section 3 will present our proposal and its impact on processor design. Our test methodology and reference processors will be exposed in section 4, with simulation results shown in section 5. Section 6 will bring our conclusions, the limitations of our approach, and our future directions of study.

P.-C. Yew and J. Xue (Eds.): ACSAC 2004, LNCS 3189, pp. 185–198, 2004.

2 Background and Prior Art

2.1 Parallelism

The way to higher performance in general purpose processors is through forms of parallelism, especially by trying to execute as many instructions as possible at the same time. The theoretical limits in the parallelism offered by different programs are far higher than those achieved in reality by current processors, for a variety of reasons. In any case, the available hardware resources are fixed by the processor's designer, and cannot be tailored to a particular application. They are chosen to get the best average performance. Superscalar processors, executing many instructions out of order every cycle, extract as much parallelism as possible during the execution of a program. This leads to complex designs, but these optimizations don't require changes to the software. In an attempt to soften the restrictions of fixed hardware resources, configurable hardware has been examined.

2.2 Reconfigurable Functional Units

Given the limitations of a fixed set of hardware resources, much research has focused on adding some reconfigurability to a general-purpose system, usually based on FPGA technology. FPGAs are most efficient for code with simple control and large data parallelism (e.g., [2]).

One can distinguish three different approaches, each bringing closer integration with the processor, and thus more generality, at the expense of performance. The first and second couple an FPGA and a normal processor, and distribute the computing tasks according to what each can do best, the difference being whether to integrate the FPGA onto the processor chip or not. There is little automation possible, and selection and coding for the FPGA must be done by hand, including the needed communication and synchronization with the processor. With well chosen applications, the gains in performance can be of several orders of magnitude [15]. In a single-chip solution, some automation is possible, usually with a smaller increase in performance than if optimizations are performed by hand (e.g., [13]).

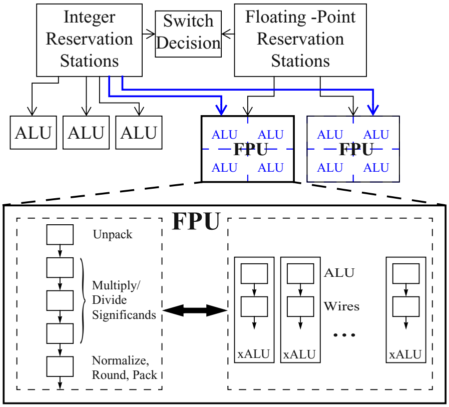
The last, most tightly coupled solution is to define the configurable logic as simply an extra *functional unit* (FU) of the processor. This reconfigurable functional unit can hold several instructions or sequences of instructions, that can be provided by a special compiler, and loaded by the processor when needed. Attempts to automate the process exist (e.g., [1], [16]), with gains similar to the second solution above (e.g., [5]). In each of these cases, the approach is to couple an existing FPGA-style block with a processor, in a more or less tightly coupled way.

We propose to consider configuration possibilities as an issue in the design of the processor's functional units, instead of adding a block of existing fully reconfigurable logic (such as FPGA technology) and trying to have the two cooperate. This implies a reduction in the configurability available, albeit with a significant gain in speed, which we hope to leverage.

2.3 Binary Compatibility

The issue of binary compatibility, ensuring that all code written for previous versions of a processor family will work on the newest model, is a complex one. However, it limits the innovation that can be implemented in the processor, since no completely novel approach may be used. As a solution to this problem, dynamic binary translation has been proposed. It aims to transform code for one architecture into another in real-time during the execution of the program. Several research projects exist [14], with one commercial implementation [8].

Our aim is to increase performance while avoiding code changes or having a major impact on timing. The lack of code changes allows our improvements to apply to all existing code and the re-use of all compiler achievements. Preserving the general timing will avoid breaking or severely limiting the performance of existing programs not suited to our modifications.



a) FPU configuration b) xALU configuration

Fig. 1. Paths between reservation stations and functional units (top), and reallocation possibilities (bottom). Each FPU can be reallocated as a number of *extra ALUs (xALUs)*. FPU operations have 5 stages, thus the FPU must be idle for 5 cycles before reallocation is possible. Likewise, the *xALUs* have 2 stages, and must all be idle for 2 cycles at the same time to allow reallocation.

3 Proposed Modification

Studies on the ideal mix and functionality of functional units in a superscalar processor have been performed [7]. These studies show that good gains can be obtained by increasing the number of identical functional units, as well as the types of instructions these units can execute. We are interested in looking for ways to reconfigure expensive functional units to perform different operations. Given the

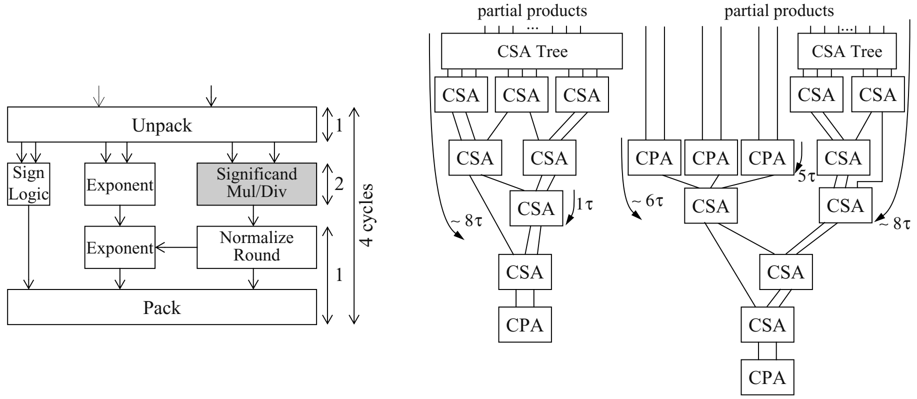


Fig. 2. Left: Structure of a floating point multiply/divide unit, with assumed cycle counts. Center and Right: Example of a 64 bit multiplier partial product reduction tree. Center: Original Wallace tree structure (total delay 14τ). Right: Proposed modification (total delay 15τ). CSAs have a delay of 1τ, CPAs have a delay of about 5τ. For clarity, the multiplexers from the Register file to the CPAs for the *xALU* configuration are not shown.

speed disadvantage of fully programmable units, which are 5 to 10 times slower than a dedicated custom logic in the same technology, we restrict ourselves to very limited changes, while maintaining speeds close to non-configurable logic.

3.1 Basic Concept

Multifunction units, such as the FPUs in the Intel Itanium 2 processor, can execute one of many different instructions each cycle. As shown in figure 1, we propose to reallocate an FPU, with a latency of 5 (figure 1a) as several *extra ALUs (xALUs)* with a latency of 2 (figure 1b). These *extra ALUs* are assumed to perform all the operations normal Arithmetic Functional Units do. Our approach differs from multifunction units since, due to these latencies, the reconfiguration decision cannot be taken on a cycle-by-cycle basis, but with a view to the next several dozen cycles. This longer view is necessary to offset the idle time before reallocation, as we have to wait for the entire functional unit to be idle before reallocating it. We trade a small decrease in speed to obtain some configurability, with the hope that adapting to applications will offset the slightly slower configurable functional units to offer a net gain in performance. We focus on a processor's floating point unit, since it is fairly large, and can often be idle during a program's execution, if the current application uses mostly integer code. Simply adding extra ALUs would further increase power consumption and area, with little impact on the results (section 5).

3.2 Standard Arithmetic Units

Current fast multipliers for fixed point numbers can be built from a tree of *Carry-Save Adders* (CSA) that adds all the partial products into two words, with a final *Carry-Propagate Adder* (CPA) for the last addition [10]. The exact structure of the tree may vary to achieve better regularity, essential for good integration. A division unit can have a similar structure, if a convergence algorithm is used. This would lead to the common implementation of a Mul/Div unit [11], with the tree structure qualitatively as in figure 2 (center). Each CSA or CPA block might need an inverter to allow subtraction. The CSA tree has $\lceil \log_{3/2}(64) \rceil = 9$ levels.

A floating-point Mul/Div unit is essentially a fixed-point Mul/Div unit with some extra logic to unpack the operands, perform Booth recoding if it is used, normalize the result and re-pack it into floating-point notation, as shown in figure 2 (left). The presence of a full CPA adder allows the re-use of the unpack and pack logic to include all floating point operations in the unit. It is also possible to use the floating point unit for integer multiplication and division, as in the Intel Itanium 2 processor [9].

3.3 Dynamic Functional Units

We propose to use the adders in an FPU as a number of *xALUs*, with characteristics similar to normal ALUs. As a CSA cannot be used to perform a complete addition, several CSAs in the tree could be replaced by CPA adders as in figure 2 (right) with only a minimal impact on the overall critical path, area and power consumption. This figure shows the proposed modifications to the reduction tree, which affect only the steering of the data, not the logic performed on it. The CPAs directly receive some of the partial products while the other partial products go through the CSA tree to allow time for the far slower CPAs to finish execution, resulting in only a small extra delay due to the unbalancing of the tree. This requires some extra logic: to handle logic operations other than add/subtract, to bring the operands for the extra instructions that will be executed, to bypass the floating point logic, and to switch between the two different modes of execution.

3.4 Effects on Functional Unit Latencies

Our reference for instruction latencies is the Intel Itanium 2 processor, one of the fastest (and certainly the largest) existing processor [17]. This processor has a latency of 1 for all ALU operations, and a latency of 4 for all FP operations and Integer Mul/Div operations. These latencies are considered here representative of current 64-bit processors, and the functional units are fully pipelined.

In deep sub-micron technology, such as 0.13μm, wires account for about 2/3 of the delays, and the differences between 0.13μm and 0.09μm are not so important in this regard. The increase in wiring to reach the *xALUs* is estimated at about double that needed for normal ALUs. Thus, if a normal ALU has a latency of 1 cycle, split as 1/3 gates and 2/3 wires, doubling the wires gives a *xALU* latency of 5/3. Taking the multiplexers to select the adders in the FPU into account, a conservative estimate for the latency of all *extra ALU* units is to double the latency of normal ALUs, for a latency of 2. As confirming this timing would require designing the entire functional core of a superscalar processor, a complex task beyond our means, simulations with a very conservative latency of 3, where about 89% of the delay is in the wires, have also been performed. Additionally, some of the bypass paths necessary to keep the pipeline as full as possible, and counted in the above calculations, are likely to already be present in the multiplier's tree linking the *xALUs* together. This also means that the overhead is less than that of simply adding extra ALUs to the processor.

The latency of the entire FPU being 4, we consider that the unpack stage takes one cycle, the multiplier tree takes 2 cycles, and the normalization and pack take the last cycle (figure 2 left). The replacement of some of the CSA adders by CPA adders will increase the total delay of the multiplier tree. From [10], considering that a CSA has a delay of 1τ, a delay of 5τ for a 64-bit CPA can be derived. The total delay for a 64-bit CSA tree with 9 levels (see section 3.2) and the final CPA, is thus $9\tau + 5\tau = 14\tau$ (figure 2 center). We assume this delay represents 2 cycles (figure 2 left), as both real processor data [9] and arithmetic considerations [11] suggest. As shown in figure 2 (right), implementing our modifications on the FPU to embed 3 CPAs in the compressor tree would increase its delay to $8\tau + 2\tau + 5\tau = 15\tau$ plus the delay a multiplexer in front of each CPA (figure 2 right). To be on the conservative side, and since functional unit latencies must be integral, we have assumed the total delay of the modified tree to be 21τ, equivalent to 3 cycles, an increase of 50%, or one cycle, for a total delay of 5 cycles in the functional unit. This adds a margin of 6τ, almost 40%, that is, many layers of logic, to the timing of the FPUs CSA tree. In any case, the partial products reduction tree is a logarithmic tree which can be easily unbalanced as needed to hide the delay of the CPAs, and so the inaccuracy due to the delays of the multiplexors and the bypass paths should not be significant overall.

Since the reconfiguration is achieved by switching the inputs of a few multiplexers, it takes only a single cycle, in addition to having to wait for the functional units to be idle, with no changes to the pipeline except the activation of the forwarding paths discussed above. The routing of the processor core must be redone to take the new data paths into account, but this kind of work must be done for newer technologies in any case. These numbers are summarized in table 1.

3.5 Switch Decision Mechanism

Given the possibility of changing an FPU into a number of xALUs, the issue of deciding when to perform this change, and when to change back, is posed. Since this decision cannot be taken every cycle because it is a global decision affecting several functional units (figure 1), an algorithm to adapt the resources to the code running at a given moment is needed. The basis for the decision is the type of instructions in the reservation stations. This gives a measure of the type of instructions the processor can expect to be executing a few cycles later. In the simplest case, the number of instructions of each type are then compared to the number of available functional units of the same type to make a decision. A *switch* is decided when the difference between the proportion of instructions of a type in the reservation stations and the resources of that type becomes too large. In the relatively common case that an instruction type should appear very infrequently, such as an integer program with very few multiplications, the algorithm above will not trigger a *switch*, since the threshold is not reached by a single instruction. In this case, we must detect that an instruction cannot be executed due to the absence of the correct resource type, and force a *switch*, regardless of the contents of the reservation stations. In all cases, a *switch decision* must wait until the functional unit(s) it wants to reallocate are completely idle, in which case it takes only a single cycle. It would be possible to *switch* while the FPU is still finishing the last calculation, during the normalization/pack stage, but this would greatly increase the complexity of the control path without a great effect on performance, through the pipelining of the *switch* logic and extra complexity in the pipeline.

3.6 Additional Considerations

The act of *switching* one or more FPUs into a number of *xALUs* increases the pressure on the memory system, as well as providing the need for extra issue, dispatch and commit width. Though the memory bandwidth remains the same, a higher number of Load/Store units are required to avoid stalling the processor due to many memory requests. In our simulations, 4 such units (as in the Itanium 2) were a good balance between performance and complexity. The widest issue rate in current processors is 8 instructions per cycle [17]. A larger issue rate increased the gains of dynamic reconfiguration, but only slightly. Thus, the issue and dispatch widths were kept at 8. The commit width need not be as large as the issue/dispatch width, since the average number of instructions committed per cycle is lower than the maximum. In our simulations, the highest average IPC was slightly below 4 (*vortex*), leading to a commit width of 8 to avoid limiting performance, as the simulator used requires it to be a power of 2, although a value of 4 could be considered.

4 Experimental Methodology

All the results presented in section 5 were obtained through the use of the Simplescalar tool set [3]. The models used for the hardware are detailed in section 4.2. On the software side, the SPEC CPU2000 [6] benchmarks were used for all tests.

Table 1. Processor model resources. The *baseline mainstream* and *baseline top* processors were compared to their *dynamic* counterparts in all simulations, with *original mainstream* and *original top* shown as references. *supertop* is equivalent to *dynamic top* with 4 additional ALUs and no reconfiguration.

Model	#ALUs (latency)	#FPUs (latency)	#Load/Store units	#xALUs per FPU (latency)	issue-dispatch-commit widths
original mainstream	3 (1)	2 (4)	2	-	4 - 4 - 4
original top	6 (1)	2 (4)	4	-	8 - 8 - 8
baseline mainstream	3 (1)	2 (4)	4	-	8 - 8 - 8
baseline top	6 (1)	2 (4)	5	-	12 - 12 - 8
dynamic mainstream	3 (1)	2 (5)	4	4 (2)	8 - 8 - 8
dynamic top	6 (1)	2 (5)	5	4 (2)	12 - 12 - 8
supertop	10(1)	2 (5)	5	-	12 - 12 - 8

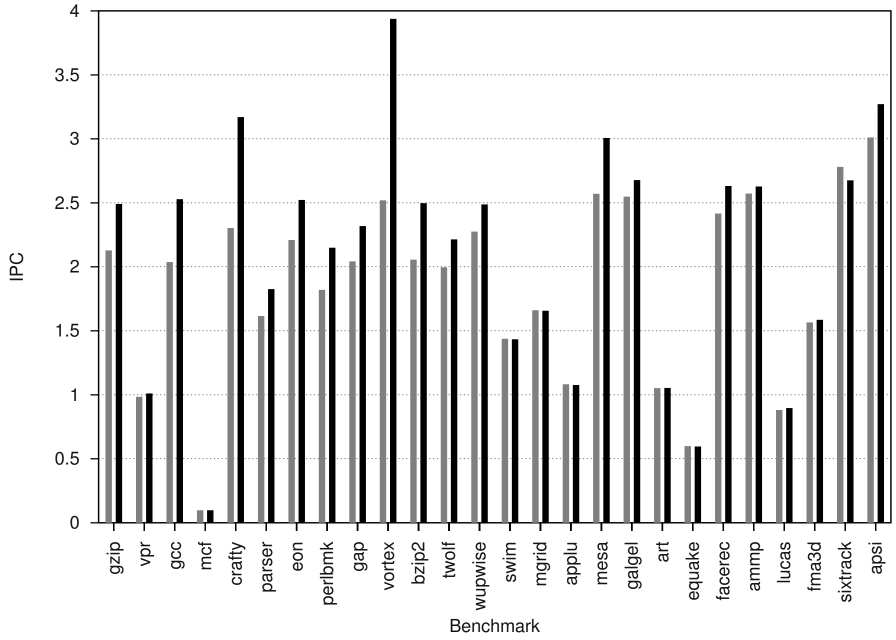


Fig. 3. Simulation results for the SPEC benchmarks for the *baseline mainstream* (light) and *dynamic mainstream* (dark) processors. There are large variations in the overall IPC, with some significant gains by the *dynamic* model.

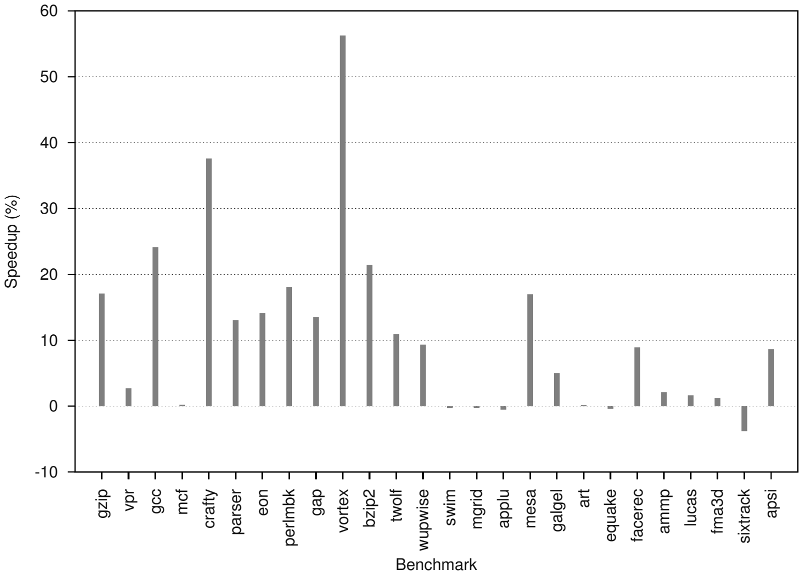


Fig. 4. Speedups between the *baseline mainstream* and the *dynamic mainstream* models. The integer benchmarks show universal gains, whereas the FP benchmark results are more varied. Except for *sixtrack*, all negative speedups are very small, less than 1% slower than the *baseline*.

4.1 Modifications to Simplescalar

The most accurate simulator in the Simplescalar tool set, `sim-outorder`, was modified so that a number of FPUs can be turned into several *xALUs*. The *switch decision algorithm* was also added to the simulator's main loop, to choose whether and how to change the allocation of resources during program execution.

4.2 Reference Processors and Models

Two different references, loosely inspired from mainstream and top server processors available today, and considered representative of the state of the art in general-purpose processors, were used:

Our *mainstream* reference is similar to the IBM Power4 processor (a single core), and is close to the average resource configuration of current processors. Each core is a 4-way superscalar processor, and has 2 ALUs, 2 load/store units, one branch unit and 2 FPUs.

Our *top* reference is loosely based on the Intel Itanium 2 processor, one of the fastest server processors available today, as measured by SPEC benchmarks. It has 2 ALUs, 4 load/store units that can also perform ALU operations, 3 branch units, and 2 floating point units that also take care of integer multiplication. Although it is a VLIW processor, its resources represent well the most aggressive configuration achievable nowadays.

For a fair comparison, both reference models are given the same memory access bandwidth and ports as our proposed model (4 or 5 load/store units and a 128-bit wide access to memory), as well as the same issue/dispatch/commit widths, giving us our *baseline mainstream* and *baseline top* models. Although these models are somewhat unbalanced, not increasing the number of load/store units would cripple the *dynamic* models, which are obtained by increasing the FPU latency as explained in section 3.4 and adding dynamic reallocation. *Supertop* is defined as a fully static top, with 4 additional ALUs and no reconfiguration, and is used to show the small difference in performance compared to the *dynamic top*. These characteristics are summarized in table 1.

4.3 SPEC CPU 2000 Benchmarks

All our tests considered the entire set of 26 benchmarks comprising the SPEC CPU2000 suite. The binaries are provided for the DEC Alpha [4] Instruction Set Architecture (ISA) on the Simplescalar WWW site [3], and have been compiled using the *'peak'* configuration. The data sets chosen are the *reference* sets from the SPEC suite. given the length of the full simulations, early *Simpoints* [12] were used to provide statistically significant results for the *mainstream* model, detailed in figures 3 and 4. Due to time constraints, and since they are only intended to show the limits of reallocation, the *top* and *supertop* models were simulated skipping a smaller number of instructions than *Simpoint* suggests. Although the individual results may vary, the average over the 26 benchmarks is similar to that obtained using *early simpoints*, and sufficient to show a trend of diminishing returns.

5 Results

5.1 Performance Results

Figure 3 shows the results of our simulations for the *mainstream* model, using the configurations in table 1, lines 3 and 5. The speedups when using perfect memories, not shown, show little difference with those presented here, demonstrating that reasonable memory latencies have little effect on the gains made by dynamic reallocation. The best performing benchmark was *vortex*, with a gain of 56%, since it uses many independent ALU operations and very few FP instructions, thus being able to make good use of the *xALUs*, and the worst was *sixtrack*, with a loss of 3.8%, which is mostly composed of FP add and multiply, and is thus strongly affected by the increase in FPU latency. The average gain for the integer benchmarks was 19%, and 3.5% for the floating-point benchmarks. The overall average for the entire suite was a gain of a little more than 10%. For clarity, the corresponding speedups for the entire set of benchmarks are shown in figure 4. There is a systematic gain, only seldom insignificant, and the rare losses in heavily FP-oriented benchmarks are rather small, with the exception of *sixtrack*.

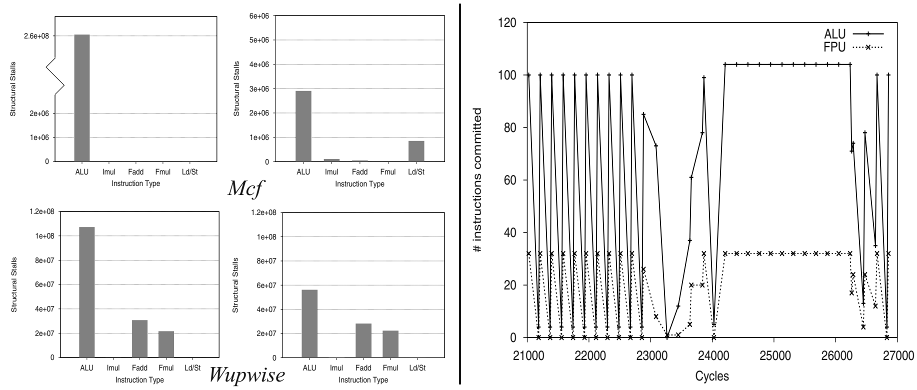


Fig. 5. Left: Structural stalls for *mcf* (top) and *wupwise* (bottom). The left side is the *mainstream baseline* case, the right side is with *dynamic reallocation*. *Mcf* is limited by ALU instructions, and shows a large reduction in ALU stalls. *Wupwise* sees little change in stalls, and thus cannot benefit from reallocation. Right: Instruction types for *galgel*. As there is no region with few FP instructions and many ALU requests, the allocation decision is to have no *xALUs*, resulting in lower performance.

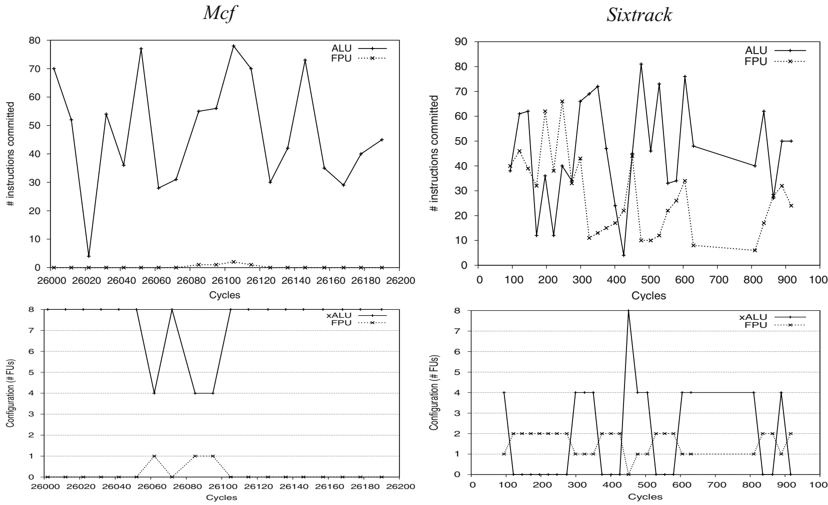


Fig. 6. Instruction types (top) and resource allocation (bottom) for *mcf* (left) and *sixtrack* (right). For *mcf*, as there are almost no FPU instructions, the configuration is always to use 8 *xALUs*. When an FPU instruction arrives, the FPU is switched to execute it, and then immediately switches back. In the case of *sixtrack*, the allocation of the FPU's resources adapts to the instruction types: when there are few FPU instructions, the units will be reallocated as *xALUs*.

The results for the *top* model, described by lines 4 and 6 in table 1, show a reduction in the gains obtained, due to far less usage of the *xALUs*, as there are already 6 ALUs in the processor. Again, memory latency did not significantly affect the speedups. The average gains were 3.7% for integer benchmarks, and 1.5% for floating-point, giving a total average gain of 2.5%. For comparison, the *Supertop* model gives an average gain of 3.1% versus the *baseline top*, at the cost of a larger set of functional units and resources on the die. If the *xALUs* latency is increased to 3, the results show a reduction in the average gain from 10% to 7%, and in the maximum gain from 56% to 35%. Thus, although this delay is somehow critical to our gain, the benefit of our system does not fully rely on these timing assumptions. Losses are not affected, since these benchmarks rarely use the *xALUs*, if ever.

5.2 Influence of Instruction Types

The large differences in speedups for the different benchmarks can be explained by looking at the instruction types used in these benchmarks. We shall use three benchmarks to illustrate this point: *mcf*, *wupwise* and *galgel*. The following graphs show good examples of the different behaviors reallocation produces. However, these are not necessarily representative of the overall benchmark results. Figure 5 (left) shows the number of structural stalls—i.e., the number of instructions of each type which had all operands ready, but couldn't execute due to a lack of functional unit, for the first two benchmarks with the *mainstream* model. The former, *mcf*, is limited here almost only by ALU instructions in addition to memory accesses, and thus benefits greatly from our proposal, since both FPUs get reallocated into many *xALUs*, switching back regularly to service the FP operations. This behavior is shown in figure 6 (left). The limitation by the Load/Store units appears because all ALU instructions that were previously waiting for a functional unit have been executed by one of the *xALUs*, and the memory accesses that had time to execute in the *baseline* case now stall the processor while waiting for the Load/Store units, which are now far less numerous than the ALUs. On the other hand, *wupwise* uses a fairly diverse mix of instruction types, with a heavy emphasis on floating-point add and multiply/divide instructions. The *switching mechanism* is constantly reallocating the functional units to try to match the instruction mix at each moment in time. In this case, the *extra ALUs* available at some moments cannot compensate for the slowdown of the FPUs' mul/div units and the delays in switching between the two. To illustrate this, a short trace of the instruction types for galgel is shown in figure 5 (right). The corresponding *switch decision*, not shown, is to never use the *xALUs*, leading to a loss in performance due to a longer latency in the FPU.

5.3 Switching Dynamics

For the resource reallocation to work, the *switch mechanism* must configure the hardware to make the best use of the configurable resources. Figure 6 (right) shows a short trace from the *sixtrack* benchmark, taken after approximately

10^9 instructions. Figure 6 (top right) displays the number of instructions committed from the ALUs and the FPUs, while figure 6 (bottom right) shows the configuration of the FPU over the same period of time.

The pattern shown is one of the startup loops in the application, and repeats regularly around the instruction count shown. At around 200 cycles, there are more FPU instructions than ALU ones, and the switching mechanism does not allocate any *xALUs*. However, at 300 cycles, the situation reverses, and one FPU is converted into 4 *xALUs*. A sharp spike in ALU instructions coupled with a sharp drop in FP instructions at 450 cycles will cause both FPUs to be reallocated as 8 *xALUs* for a brief moment, before resuming FP functions. A long period of relative stability, between 650 and 850 cycles leads to a unchanging configuration.

6 Conclusions and Future Work

We have proposed a method to gain some hardware adaptability to the code running on a general-purpose processor that does not sacrifice the speed of the configurable unit or compromise binary compatibility. This technique is distinctive in requiring the logic of the superscalar processor to make more global decisions than it normally does. The conditions for the simulations have been derived from real data measured from 0.13μm technology. The results show the use of a dynamic FPU is quite interesting in the case of processors with a modest number of ALUs, and that naturally the interest declines with a large number of ALUs already in the processor. Our idea, based on giving the processor more possibilities for parallelism, should be seen as an example of the possibilities in superscalar processors that can be exploited by multi-cycle reallocation decisions. When superscalar processors will enter the embedded System-on-Chip world, the common use of domain-specific instructions or coprocessors for these applications will increase the opportunities for similar forms of reconfiguration.

We intend to apply control theory to the decision mechanism, in order to better tailor the resources to the application. Simulations on a SMT processor are expected to produce interesting results, due to the extra parallelism exposed by the multiple threads. We also envision to research the possibility of using software hints in the code to guide resource reallocation. While this would maintain backward binary compatibility, it will require a recompilation and some analysis of the code to produce better gains. In a similar vein, it might also be possible to apply this method to VLIW processors, in which case the resource allocation would simply be another information generated by the compiler.

Acknowledgment. We would like to thank the anonymous reviewers for their insightful comments.

References

1. K. Atasu, L. Pozzi, P. Ienne, *Automatic Application-Specific Instruction-Set Extensions under Microarchitectural Constraints*, Proc. of the 40th Design Automation Conference, June 2003.
2. M. Borgatti et al., *A Reconfigurable Signal Processing IC with embedded FPGA and Multi-Port Flash Memory*, Proc. of the 40th Design Automation Conference, June 2003.
3. D. Burger, T. M. Austin, *The Simplescalar Tool Set, Version 2.0*, www.simplescalar.com
4. J. H. Edmondson, et al., *Internal organization of the Alpha 21164, a 300-MHz 64-bit quad-issue CMOS RISC microprocessor*, Digital Technical Journal, 1995.
5. S. Hauck, T. W. Fry, M. M. Hosler, J. P. Kao, *The Chimaera Reconfigurable Functional Unit*, IEEE Symposium on Field-Programmable Custom Computing Machines, 1997.
6. J. L. Henning, *SPEC CPU2000: Measuring CPU Performance in the New Millennium*, IEEE COMPUTER, July 2000.
7. S. Jourdan, P. Sainrat, D. Litaize, *Exploring Configurations of Functional Units in Out-of-Order Superscalar Processors*, Proc. 22nd Annual Int'l Symposium on Computer Architecture, June 1995.
8. A. Klaiber, *The technology behind Crusoe processors*, Transmeta Corporation, Jan. 2000.
9. C. McNairy, D. Soltis, *Itanium 2 Processor Microarchitecture*, IEEE Micro, March 2003.
10. A. R. Omondi, *Computer Arithmetic Systems: Algorithms, Architecture and Implementations*, Prentice Hall, 1994.
11. B. Parhami, *Computer Arithmetic Algorithms and Hardware Designs*, Oxford University Press, 2000.
12. E. Perelman, G. Hamerly, B. Calder, *Picking Statistically Valid and Early Simulation Points*, International Conference on Parallel Architectures and Compilation Techniques, September 2003.
13. R. Razdan, M. D. Smith, *A High-Performance Microarchitecture with Hardware-Programmable Functional Units*, Proc. of MICRO-27, Nov. 1994.
14. G. T. Sullivan, D. L. Bruening, I. Baron, T. Garnett, S. Amarasinghe, *Dynamic Native Optimization of Interpreters*, IVME 03, June 2003.
15. R. D. Wittig, *OneChip: An FPGA Processor With Reconfigurable Logic*, IEEE Symposium on FPGAs for Custom Computing Machines, 1995.
16. Z. A. Ye, N. Shenoy, P. Banerjee, *A C Compiler for a Processor with a Reconfigurable Functional Unit*, ACM Int'l Symposium on Field Programmable Gate Arrays, 2000.
17. In-Stat/MDR Workstation and Server Processor Chart, `http://www.mdronline.com/mpr/cw/cw_wks.html`

Multiple-Dimension Scalable Adaptive Stream Architecture[1]

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

Computer School, National University of Defense Technology
Chang Sha, Hu Nan, P. R. of China, 410073
wxxwm@263.sina.com

Abstract. Intensive processing applications, such as scientific computation, signal processing, and graphics rendering, motivate new processor architectures that place new burdens on the designer. These applications named Stream Applications demand very high arithmetic rates and data bandwidth, but lack data reuse. At present modern VLSI technology makes arithmetic units relatively cheaper. MASA(Multiple-dimension scalable Adaptive Stream Architecture) presented in this paper is a prototype that operate on streams directly. It is different from DSP and special high performance single-chip architecture because it combines flexibility and high performance. It has basic features of all stream processing, provides bandwidth hierarchy, makes ALU array execute with full loads and decomposes application into a set of computation modules to execute space-multiplexing or time-multiplexing. The multiple dimensions scalability of MASA, includes task-level, loop-level, instruction-level and data-level, and enables it to meet the demand of stream applications. This paper describes MASA architecture and stream model in the first half, and explores the features and advantages of MASA through mapping stream applications to hardware in the second half.

1 Overview

Under the power of Moore's law, the number of transistors integrated on chips has been increasing rapidly and the performance of chips has been enhanced constantly. In a contemporary 0.15μm CMOS technology, a 32-bit integer adder requires less than 0.05mm^2 of chip area. Integrating more ALUs on single chip is not a problem any more. On the other hand this situation brings some new problems to the computer designer. One is how to support so many ALUs with enough instruction and data. The long wire delay is becoming more and more unavoidable owing to the increase of the density on chip. The other is how to make full use of the ability of chips' integration, that is, how to make full use of chips' area to compute other than traditional general-purpose architecture, for example only 6.5% of the Itanium 2 die is devoted to arithmetic units[1], and large fraction of its die area is consumed by other processing such as data cache, branch prediction, out-of-order execution and communication

[1] This work was supported by the 973 Project(5131202) and the 863 Project(2001AA111050) of China.

P.-C. Yew and J. Xue (Eds.): ACSAC 2004, LNCS 3189, pp. 199–211, 2004.

schedule. In addition, low efficiency is worth attention as the demands of applications increase.

Under this condition, traditional microarchitecture has met the challenge. How is the performance of microprocessors kept on scaling at the rate of 50% or even more per year[2]? Many researches on new microarchitecture of billion or more transistor chips are going on very well. And some technologies have come to life, such as systolic arrays in 1970s, data flow in 1980s and vectors in 1990s.

Stream architecture operates on data streams. It is first applied in media processing because of intensive computation, high data parallelism, and little data reuse. Stream has various formats, fixed length or variable length, and supports for collections of complex or simple elements. But data stream is consistent with the colloquial sense. According to Webster, a stream is “an unbroken flow(as of gas or particles of matter)”,a “steady succession (as of words or events)”,a “constantly renewed supply,” or “a continuous moving procession (a stream of traffic)”[3]. The difference between stream and data flow is that some architecture adopts instruction driven rather than data driven. Though there are several kinds of stream architectures, the common feature is to take stream as architectural primitives in hardware. Stream architecture is suitable for VLSI technology and supports enough functional units to achieve the needed arithmetic rates because it has scale register file architecture and so on. At the same time its organization is quite simple, so it will be a hotspot in the near future.

MASA (Multiple-dimension Scalable Adaptive Stream Architecture) is a kind of stream architecture. The definition of programming is expanded, meaning that almost all parts are programmable, including inter-ALU switch, inter-arithmetic pages communication and register organization. So it has perfect flexibility compared to special architecture. It shares common characteristics of all stream processing, provides multiple-level bandwidth hierarchy, make ALU arrays execute with full loads. According to the demands of various applications, MASA decomposes application into some modules to do space multiplexing or time multiplexing and can be scaled in multiple dimensions (including task-level, loop-level, instruction-level and data-level).

The remainder of this paper is organized as follows. Section 2(“related work”) cites prior work in streaming and architecture that has inspired MASA. Section 3(“MASA microarchitecture”) presents MASA. Section 4(“the MASA’s stream model”) discusses executing model of stream in MASA. Section 5(“stream application studies”) analyses the computing process with mapping a typical application onto the stream executing model of MASA. Compared with processing model of scalar and vector, it explores the features and advantages of MASA. The last section (Section 6) summarizes the conclusions drawn in this paper.

2 Related Work

MASA draws heavily on the prior work of numerous parallel models and architectures. This section highlights only a few of those works.

Viram[4,5] adopts multiple-level vector pipeline and places large memory in the chip. It integrates typical vector with PIM technology. However, it has limited scalability.

Raw[6], as a typical representative of tiled architecture, implements thread-level parallelism and is reconfigurable. But the connection of inter-tile is complex and the cost of crossbar is very significant.

Score[7,8] exploits thread-level parallelism like Raw, and has good compatibility by hiding the size of hardware from programmer. Although computer pages are reconfigurable, the waste of logic is quite large and the implementation is hard.

Imagine[9,10,11], as a pioneer of stream architecture, expands vector technology. The structure is very simple. It provides bandwidth hierarchy and resolves the bottleneck very well. It performs time multiplexing for multiple kernels in single chip and implements instruction-level and data-level parallelism, but task-level parallelism is weak. On the basis of Imagine, a supercomputer called Merrimac[19] is developed. One Merrimac stream processor is very similar to Imagine. MASA is influenced by Imagine heavily.

VLIW[12] exploits instruction level parallelism well. A VLIW architecture is low power efficiency, because in this case, a compiler performs scheduling of operations rather than hardware in superscalar architecture. But the scalability is very poor. At present many improved VLIW have come out such as dynamic VLIW technology [13].

Trips[14,15] is a general-purpose chip facing 2010 which consists of one or more inconnected grid processors working in parallel. It is a Polymorphous system, meaning that hardware should adapt itself to a variety of runtime workloads. However, it is essentially a reconfigurable array, just that the task of compiler and OS is new. Anyhow, it is quite complex.

3 MASA Microarchitecture

The prototype microarchitecture of MASA is shown in Figure 1. MASA is a programmable stream processor, which works as a stream coprocessor. Scalar program is executed on the host processor. Actually, MASA implements logic stream model in a single chip. The whole architecture can be divided into three layers by different bandwidth levels, and each layer owns its special controller, communication style and storage unit.

In the outer layer, scalar processor executes scalar instructions like MIPS and transfers explicit stream instructions to instruction caches in each *compute engine* (CE), which is a set of units executing thread-level tasks partitioned ahead. *Compute engine* can complete a task independently, so that MASA may own only one *compute engine,* adding the engine only when more task level computing is required. Stream memory system transfers streams between *stream register file* (SRF) and off-chip SRAM or SDRAM. MMU and *compute engine* are connected with multiple buses that we call *Multi Stream-bus*. Engines get stream-bus ownership by asynchronous request-answer signal's competition. After request is granted, *stream records* are transferred to SRF in the form of group or burst sequentially. *Stream record* is the elementary particle of data block. *Arbiter* is responsible for the grant or other signals, and the Arbiter principle can be either time-multiplexing or dynamic priority. Use priority for each bus can be set dynamically, that means the priority may be modified by the host processor. For each bus, the Arbiter has a special register to indicate the priority. Because of continuity of stream data, the bus interface of each engine has a lot of buffers called *stream buffers* in order to hide the latency of memory access.

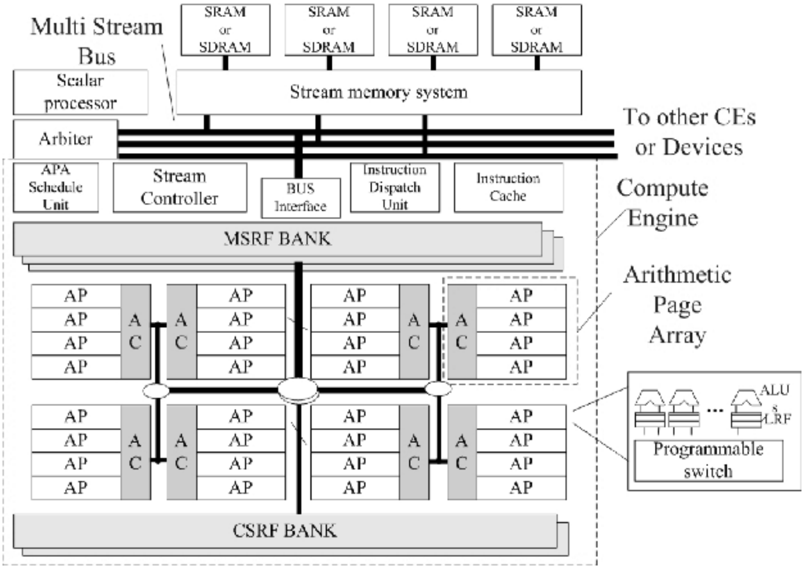


Fig. 1. Hypothetical, single-chip MASA system

The next layer is *compute engine*. Each engine consists of many *arithmetic page arrays* (APAs), SRF banks and a control unit. *Arithmetic page array*, which contains lots of arithmetic logic units, is the basic unit for executing kernel microcode program. APA schedule unit determines which APA executes a certain kernel, that means how to map the logic kernel to physical execute units. There is a scoreboard making records about the information of APA's status in the schedule unit. According to the information from schedule unit, *stream controller* dispatches stream instructions to one or more APAs. Scalar processor controls stream controller by stream instructions. These instructions determine kernels' process procedure and which APA is distributed to which kernel. Kernels are space-multiplexed or time-multiplexed that will be discussed in the next section. Instructions are combined as VLIW, and microcode consists of a series of 256~512 bit instruction words. When microcode is fetched from memory, it will be dynamically reorganized by an instruction issue unit to adapt to the scalability. Since instruction buffer of APA is limited and instruction will be regularly reused, an instruction cache is necessary, and the basic access unit is no longer an instruction but the whole microcode of a kernel. Instructions in the next execution that have been reorganized last time are fetched from instruction cache, so we needn't reorganize or distribute them again, otherwise it will be sent to off-chip memory instead. All transfers of data are passed through SRF. Stream data represents the principle of locality, so SRF is divided into several *banks*, whose size is tens Kilo-words. So that MASA can effectively increase the SRF's bandwidth. Data in a logic kernel is not allowed to store across banks. According to the destination of data, SRF may partition into main SRF (*MSRF*) banks and communication SRF (*CSRF*) banks. The former is responsible for the exchange of data between memory and engine, or dataflow between kernels. And the latter is responsible for the communication of global data among computer engines. MASA uses topology like tree to connect SRF to APA, which follows weak dependence and unilateral transformation of stream. At this layer, both instruction and data are transformed in the same data path, but in different time, so the execution of program is divided into two parts: configuration time and execution time. Partition of kernels and issue of instructions are completed in configuration time. After that, stream controller answers for transferring data and executing program according to kernels divided ahead. During this period, kernel program is kept unchangeable in APA and data streams are transformed from SRF to APA in sequence.

The most inner of MASA is kernel layer. Microcodes are executed in APA in the form of SIMD. Each APA consists of many *arithmetic pages* (APs) and one central *array controller*. And each AP contains eight to sixteen arithmetic logic units, including floating adder, floating multiplier, divider and some special function units. Each ALU owns its *local register file* (LRF) to keep its local data. A LRF is keeping a small suitable size that can be accessed fast enough. All ALUs in an AP are connected by programmable switch net, so ALUs in an AP are able to communicate each other fast and flexibly. But the communication between APs will take much more cycles; fortunately the stream application limits these communications in a negligible degree. VLIW codes of kernel program are buffered in array controller. AC has special instruction RAM to hold thousands of VLIW, and that is also responsible for decoding packed VLIW to control signals and transferring these signals to every AP in an APA.

The bandwidth hierarchy of stream processor insures kernel program not to access memory directly. The memory access latency can be effectively hidden by buffer and soft pipeline etc. All inputs and outputs of kernels have to transfer through SRF as the type of stream. Analogously, temporary results and local data that just exist or are used in a kernel are limited in the fast LRF, so the bandwidth requirement for SRF can be greatly reduced. By the way, operands' fetch time is explicitly determined so that no miss will happen. As a result, in kernel level all instructions' executing time are explicitly determined without any potential uncertain cycles, so in the control unit we can get an accurate static VLIW schedule time table. For this static structure, VLIW could be most efficient because it simplifies or cancels the hardware unit for some dynamic schedule such as extra register renaming, dependence detection, issue out of order and so on. Obviously, according to the features of stream algorithms, MASA emphatically solves the problem of high bandwidth and multi-kernel executing, so it can achieve very fast speed but simple structure which are of great benefit to reduce area and power efficiency.

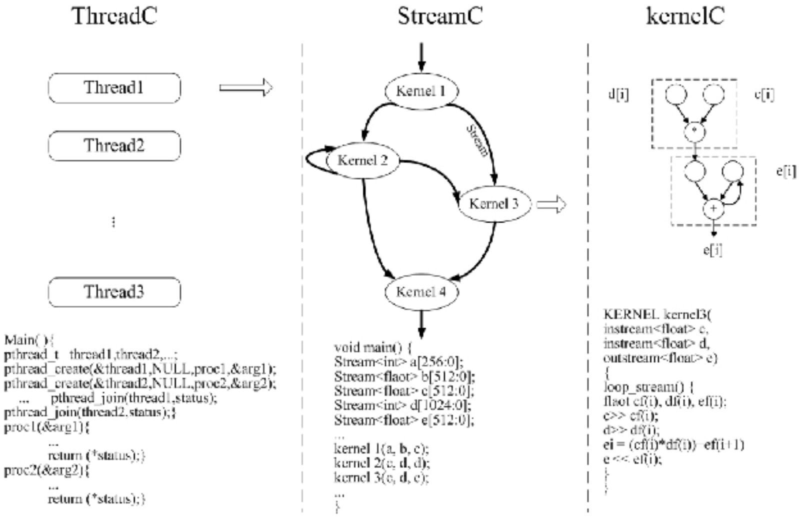
According to bandwidth hierarchy, MASA defines different layers of architecture. In fact these layers correspond with different parallelisms that are task-level-parallelism (TLP), instruction-level- parallelism (ILP) and data-level-parallelism (DLP) strictly, which make us easily scale MASA in any dimension of these parallelisms. It means MASA can be emphatically scaled in the most needed dimension, so MASA's scalability is much more powerful and efficient. However, the most challenging work in scaling is how to partition basic APAs to different kernels in balance. MASA could solve this problem by the explicit definition in program or the cooperation of compiler and hardware (such as APA schedule unit). Therefore, MASA is well adaptive.

4 The MASA's Stream Model

The definition of stream in MASA is similar to that in other stream architectures. It consists of successive ordinal isomorphic elements. Stream has various formats, fixed or variable length, and complex or simple elements.

MASA works as a coprocessor for a scalar processor. It receives stream data and stream instructions from scalar processor, and transfers the results to the host. On the other hand, scalar processor performs scalar programs.

The application of stream on MASA is divided into three levels: stream-thread level, stream scheduling level and kernel execution level. The instance of the model is shown in Figure 2.



```
Main( ){
pthread_t   thread1,thread2,...;
pthread_create(&thread1,NULL,proc1,&arg1);
pthread_create(&thread2,NULL,proc2,&arg2);
    ...     pthread_join(thread1,status);
pthread_join(thread2,status);}
proc1(&arg1){
        ...
        return (*status);}
proc2(&arg2){
        ...
        return (*status);}
```

```
void main() {
Stream<int> a[256:0];
Stream<flaot> b[512:0];
Stream<float> c[512:0];
Stream<int> d[1024:0];
Stream<float> e[512:0];
...
kernel 1(a, b, c);
kernel 2(c, d, d);
kernel 3(c, d, e);
...
}
```

```
KERNEL kernel3(
instream<float> c,
instream<float> d,
outstream<float> e)
{
loop_stream() {
flaot cf(i), df(i), ef(i):
c>> cf(i);
d>> df(i);
ei = (cf(i)*df(i))+ef(i+1)
e << ef(i);
}
}
```

Fig. 2. Hypothetical MASA model

At the top level, the stream-thread level programming is done using thread C- like language. It is responsible for memory access scheduling in multi-buses and low-intensity communication among threads. Task is scheduled at the thread level. However, this level is not necessary, only used in large-scale computation. (For instance, there are 2 or more engines in MASA).

The second level programming is done using extended streamC[16] language. A *stream task* is decomposed to a series of computation kernels that deal with a great number of stream data. It permits determining executing order and size of kernel pattern by programmer. Program at stream scheduling level controls the whole flow of a stream task's execution, including communicating towards scalar processor, loading stream across the high-speed bus from off-chip, storing stream to off-chip memory, and starting kernel. The flow route of stream among kernels is similar to dataflow graph. The kernel operates successive data in the input stream, and produces the output stream as the input stream for other kernels. Similar to product line, each kernel just processes one record of input stream every time, and then appends result to the output stream. Kernels may be executed in two modes: time-multiplexing and space-multiplexing mode. In the former mode kernels are executed one by one at different time, and gain all the computing recourse exclusively. In the latter mode, several kernels share the whole computing recourse at the same time, we call the set of these kernels *synchronous kernels*. If adopting space-multiplexing mode, the output record will be sent to next kernel directly, as it may be transformed through buffer in time-multiplexing mode.

Computation kernels lie in the third level. It describes execution of single kernel in kernelC[16] -like language. The details of arithmetic operation and data executing mode are defined at this level. A kernel executed in AP is composed of VLIW instructions to exploit the great instruction parallelism. Moreover, by requiring programmers to explicitly use appropriate type of communication for each data element, this level expresses the application's inherent locality. All the temporary values just generate and exist within kernel. Thus the model does not use inter-kernel communication for temporary values.

Before the execution in chip, compiler and hardware decide how many APAs are available to execute several kernels simultaneously or time-sharing, and how many APAs each kernel occupies. Between APAs that implement different kernels, data is transferred in the form of stream. When forming logic kernels, programmer should consider the number of kernel's arithmetic operations and the size of intermediate results. An ideal kernel should have enough quantity of operation on local data but small set of intermediate results between kernels. At the same time, the balance between producer kernel and consumer kernel is important. We can introduce kernel fission and fusion technology into decomposing kernels.

5 Stream Application Studies - Fluid Compute

Stream applications are defined as the applications that can be mapped to stream programming model. Those focus on intensive computation domains, including scientific computation, graph rendering, media processing and so on. Features of stream application are obvious: First, computation is intensive and is limited within given time. Second, data has little reuse and weak dependence. Third, it is easy to divide program into some modules. Fluid compute is a typical scientific computation, which is applied in many important domains, such as aerospace, space flight, machine manufacture and so on. This paper takes numerical simulations of complex steady flow in hypersonic free stream as a stream application to map to MASA. We analyze kernel decomposing and intermediate result, and also compare MASA with some other architectures.

The whole computation has three steps in terms of LU-SGS: First, input the parameters and choose the format of space and time. Second, calculate the values of basic controlling equation group. Third, advance by time and iterate to solve the equations. Repeat the second and third steps till result is convergent[17]. Figure 3(a) diagrams part of LU-SGS algorithm.

The program has some characteristics shown in the following:

- Data represents locality and computation is advanced towards unilateral direction. The value of each point only needs information of two or three points around, so it can be paralleled on several points.

- The dependence distance of iteration is very small, because each iteration only needs the results of one to three latest iterations. It is convenient to exploit kernel-level parallelism by loop unrolling or software pipeline.
- There is no dependence between fluxes computation, so they can be executed in parallel as multiple tasks. The function in the program is easy to be divided into some blocks in accordance with loop. As a result, they can be mapped onto several kernels directly.
- Computationally intensive. The application requires many arithmetic operations per memory reference.
- There is few conditional branch which may be converted to conditional instruction. The characteristics of computation and data match MASA.

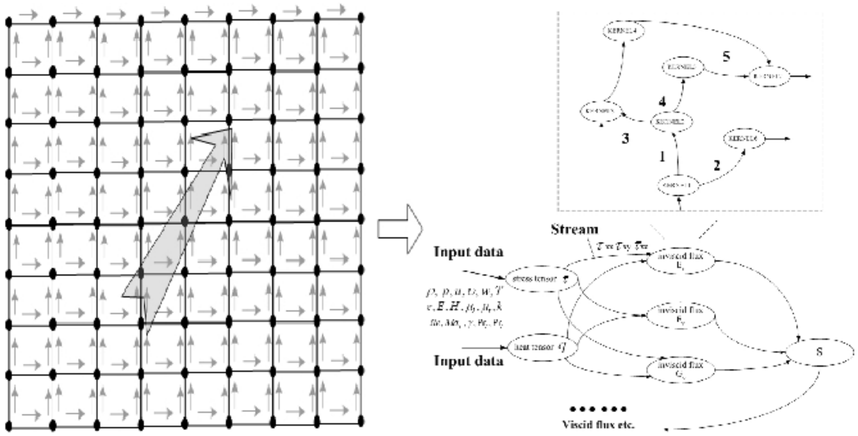


Fig. 3. (a) part of LU-SGS algorithm (b) complex steady flow computing

The main part of the computation is to calculate partial differential equation to get six fluxes, that is $\frac{\partial U}{\partial t}+\frac{\partial E}{\partial x}+\frac{\partial F}{\partial y}+\frac{\partial G}{\partial z}=\frac{\partial E_v}{\partial x}+\frac{\partial F_v}{\partial y}+\frac{\partial G_v}{\partial z}$ where U is conservative variable, E, F, G, E_v, F_v, G_v are viscous or inviscid fluxes in the direction of x, y, and z. Computation of three inviscid fluxes is most intensive. Figure 3(b) shows how we map solving partial differential equation to the stream programming model in brief. Where τ and q mean stress tensor and heat tensor respectively. H is enthalpy of unit quality. ρ , p , u , v , w , T mean density, pressure, three components of the velocity and temperature respectively. e, E, H , μ_l , μ_t , k mean energy, enthalpy and coefficient of viscosity respectively. Re , Ma_∞ , γ , $\Pr_l$, $\Pr_t$ mean Reynolds Number, Mach Number, heat ratio and Prandtl Number respectively.

Take 300 thousands points as input data, the computation of three inviscid fluxes is 1.5 to 2 billion single precision floating operations for each iteration, which mainly contains addition and multiplication. The total computation for each iteration is about 2.5to 3.5 billion. The test program is small-scale, it needs 10000 steps to acquire constringent result at least. 15000 steps are assumed, and the total amount of arithmetic operations will be 4500 billion. It takes more than twenty hours, and the cost of memory is about 300MB, when the program is run on Pentium 4.

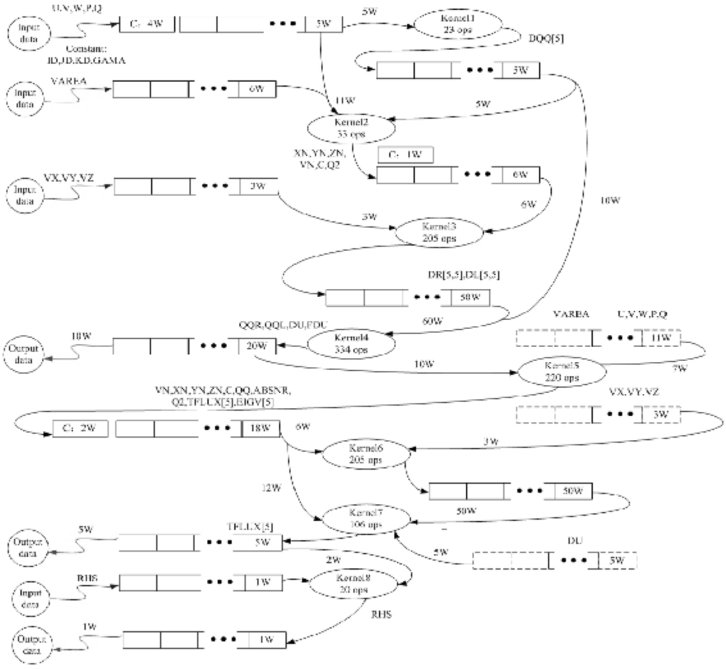


Fig. 4. Stream-Based inviscid flux computing

Figure 4 shows how we map computation of a inviscid flux to the stream programming model. The processing is composed of eight computation kernels that operate on data streams in succession. Where input data is fetched from memory and output data is stored to memory, rectangle represents stream in SRF, and a grid represents a stream element. Temporary data required by arithmetic operations is stored in LRF. For example, kernel3 receives 2 input streams and produces 1 output stream, performing 205 arithmetic Ops.

Table 1 compares the memory, global register, and local register bandwidth requirements of a stream architecture (MASA) with a vector processor and a scalar processor for the kernel3[2].

The content illuminated in the left-most column of Table 1 is the number of memory references, SRF references, and LRF references for the stream architecture. During the entire period of pipeline, the stream architecture performs 35 memory references as stated in Figure 4. Amortizing this across the eight kernels gives 4.375 memory references per kernel. The total SRF reference of the kernel3 is 59, that contains 9 words read from the SRF and 50 words written to the SRF. The 615 words of LRF bandwidth are used for each node to carry out the kernel that requires 205 arithmetic operations. The additional LRF accesses beyond the 615 come from

[2] This number is set by 8 ALUs in an AP of MASA.

register accesses to originally store data from the SRF into a local register file, and register transfers required for other kernels.

Table 1. 32-bit references per point for kernel3

	Stream	Vector	Scalar
Memory	4.375	59(13.5)	288(65.8)
Global RF	59	552(9.3)	1299(22.1)
Local RF	674	N/A	N/A

The next two columns present the number of references for the same kernel on a vector processor with an organization similar to the MASA processor. Considering vector operations are primitive arithmetic operations instead of compound stream operations, using global (vector) register file instead of local register files as a source and sink of data for the arithmetic units, and software pipeline, the vector processor requires respectively 13.5 times, 9.3 times as much memory bandwidth, global register file bandwidth as a stream processor.

The last two columns of Table 1 compare these numbers to a scalar processor by giving both the absolute number of references and the ratio to a stream processor (in parentheses). The scalar numbers were generated by compiling the kernel3 for MIPS using version 3.0.2 of the gcc compiler. The scalar processor requires respectively 65.8 times, 22.1 times as much memory bandwidth, global register file bandwidth as a stream processor. Considering that the number of memory references for the stream architecture is an average, and the kernel3 is run in the middle of pipeline, some data required would be in SRF, need not access memory, so the number of memory references for the scalar architecture is relatively magnified. However the difference is still great. Cache in the scalar processor is useful to shorten the gap, but there is little data reuse in stream application, so the benefit is limited.

MASA exploits the space parallelism that multi-kernels can run concurrently. The advantages of this feature can be appreciated by comparing it with other typical stream processors such as Imagine. Certainly, the advantage is at the cost of hardware complexity.

When SRF is full of intermediate results, the kernel which is running have to be changed to the other kernel to consume intermediate results. So each time the kernel runs, the number of records operated is limited. We call the number *kernel executing granularity* (KEG). Kernel executing granularity is limited by size of intermediate results and SRF. Table 2 compares the intermediate result size per node, maximum KEG and SRF requirement ratio of MASA with typical stream processor for inviscid flux computing. We analyze the intermediate result of each fluid node. These two processors are assumed to have the same computing ability and SRF size, though MASA's distributed SRF is different from Imagine's central SRF in bandwidth and utilization. In MASA, data between kernels running at the same time will be transferred and consumed immediately, so this procedure can generate smaller intermediate result. Therefore, smaller intermediate results could effectively increase Kernel executing granularity and utilization ratio, decrease the cost of exchanging kernels; and it also makes kernel operate on a longer stream that relieve the short stream problem.

For the same kernel executing granularity, MASA can decrease the SRF's size to a half of typical stream processor or even smaller. This advantage makes it much easier for larger scaling. Table 2 shows that kernel executing granularity is not linear with the number of synchronous kernels.

Table 2. SRF requirement for inviscid flux computing[3]

	Typical Stream architecture	MASA (2 synchronous kernels)	MASA (3 synchronous kernels)
Average intermediate result size per node	45	22.7	19
Maximum intermediate result size per node	94	52	26
Maximum KEG [4]	300s	600s	1200s
SRF size ratio at the same KEG	100%	55%	27%

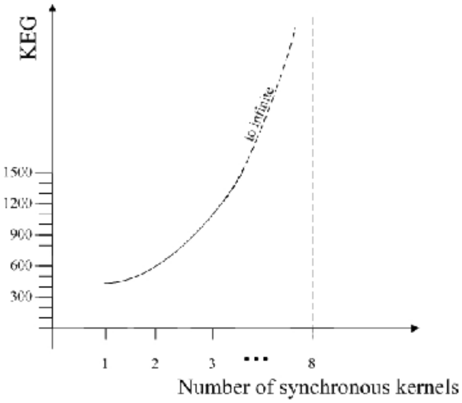


Fig. 5. The number of synchronous kernels and Kernel executing granularity for inviscid flux computing

According to different applications, as the number of synchronous kernels increases, consolidated intermediate results will decrease rapidly. For example, in table 2, typical stream processor's max intermediate result is 94W, that it is only 52W in column 2. As the number of synchronous kernels growing to 3, this value decreases to 26W. Support that when all of 8 kernels in figure 4 are running in MASA at the same time, intermediate result's size will become a constant which is irrelative with Kernel executing granularity. This means Kernel executing granularity may approach infinity in theory as figure 5 shows. However along with the number of synchronous kernels' increasing, decomposition and schedule will become more complex consequently. Also this complexity does not linearly increase and will go to

[3] Data of typical stream processor is estimated according to paper [18].

[4] 128KB SRF is assumed.

unacceptable at last. Furthermore, the load balance among kernels will be hard. In making a design trade-off, we think it better that the number of synchronous kernels should be fewer than 4 in middle-scale MASA.

6 Summary

MASA is a single-chip stream prototype architecture that supports the stream programming model by providing a data bandwidth hierarchy matched to the demands of typical computation-intensive applications. MASA supports multiple kernels running simultaneously or in sequence, dataflow among kernels organized around distributed SRF banks and all arithmetic operations are performed on streams transferred to and from the SRF.

Stream programming model exposes the parallelism and locality of stream applications in a manner that is well matched to the capabilities of modern VLSI technology. Through the analysis of above, it is known that stream application could be mapped to MASA model naturally, and a great number of ALUs on MASA will work fully. For complex steady flow in hypersonic free stream, a typical scientific computation, MASA infinitely reduces the demands for global register and memory bandwidth over a scalar processor and vector processor. This enables stream architecture to make efficient use of a large number of arithmetic units without global bandwidth becoming a bottleneck. About this, MASA is similar to other stream architectures.

Because MASA can do space multiplexing for multiple kernels, for some applications with computation-intensive in limited time, such as game system, set-top box, network switch and so on, they are easy to be mapped on MASA in order to achieve higher executing efficiency than other similar stream architectures. Another advantage is to reduce the demand of SRF size. As the number of kernels that run at the same time increases, the size of intermediate result decreases rapidly to a constant. So the runtime of the kernel each time trends to infinite ideally, kernel needs not to be changed, and it reduces the cost of kernel trembling.

MASA exposes concurrency at multiple levels: at AP level, DLP and ILP are exploited; at APA level, kernel level parallelism is exploited; at Engine level, TLP is exploited. In every dimension, MASA takes full advantage of function unit to do space multiplexing or time multiplexing for multiple kernels. The capability for adaptation is very good.

There are a lot of problems to solve remaining in the research of MASA architecture, such as the synchronous communication with scalar processor, the decomposition of multi-kernels, the dynamic scheduling algorithm, the appropriate size of register file at each level, the amount of ALUs in APA, the execution of conditional stream, the structure of communication network and the optimization of multiple-dimension scalability and so on.

Acknowledgements. The authors thank Tian Zhengyu for his knowledge of fluid compute. We also thank the reviewers for their valuable feedback.

References

1. U.J. Kapasi et al., programmable stream processor, IEEE computer, Aug 2003.
2. Bill Dally, Pat Hanrahan, and Ron Fedkiw, A Streaming Supercomputer, Whitepaper, Sep 2001.
3. H.Hoffmann et al., stream algorithms and architecture, MIT Laboratory for computer science, 2003.
4. C. E. Kozyrakis et al., Scalable Processors in the Billion-Transistors Era: IRAM, IEEE Computer, Vol 30 Issue 9, Sep 1997.
5. C. E. Kozyrakis, A Media-Enhanced Vector Architecture for Embedded Memory Systems Report No. UCB/CSD-99-1059, Jul 1999.
6. M. B. Taylor et al., The Raw Microprocessor: A Computational Fabric for Software Circuits and General Purpose Programs, IEEE Micro, 2002 ¾.
7. Score project, http://brass.cs.berkeley.edu/SCORE/
8. Eylon Caspi et al, A Streaming Multi-Threaded Model, the Third Workshop on Media and Stream Processors, in conjunction with MICRO34, Austin, Texas, Dec 2001.
9. B.khailany, W.J.Dally et al., Imagine: media processing with streams, IEEE micro, 2001.3/4
10. U.J. Kapasi,W.J.Dally et al., the Imagine stream processor, Proceedings of 2002 International Conference on Computer Design
11. Imagine project, http://cva.stanford.edu/Imagine/project/im_arch.html
12. Joseph A.Fisher, Very Long Instruction Word Architectures and the ELI-512, 25 Years ISCA: Retrospectives and Reprints 1998
13. Li Shen, The Research and Implementation on Key Issues of Dynamic VLIW Architecture, Ph.D. Thesis, Dept. of Computer Science , National University of Defense Technology, Dec 2003.
14. Trip project, http://www.cs.utexas.edu/users/cart/trips/
15. Karthikeyan Sankaralingam et al., Exploiting ILP, TLP, and DLP with the Polymorphous TRIPS architecture, 30th Annual International Symposium on Computer Architecture, May 2003.
16. Mattson, A Programming System for the Imagine Media Processor, Stanford Ph.D. Thesis, 2001
17. Zhengyu Tian, Numerical Simulations of Multiplex Unsteady Flow in Hypersonic Free Stream , Master Thesis, Dept. of Aerospace and Material Engineering, National University of Defense Technology, Dec 2003.
18. S. Rixner et al., A Bandwidth-Efficient Architecture for Media Processing, 31st Int'l Symp, Microarchitecture, IEEE Computer Society Press, 1998
19. W.J.Dally et al., Merrimac: Supercomputing with streams, SC'03, Nov 2003.

Impact of Register-Cache Bandwidth Variation on Processor Performance

Kentaro Hamayasu and Vasily G. Moshnyaga

Dept. of Electronics Engineering and Computer Science, Fukuoka University
8-19-1 Nanakuma, Jonan-ku, Fukuoka 814-0180, JAPAN
{hamayasu, vasily}@vlab.tl.fukuoka-u.ac.jp

Abstract. Modern general-purpose processors employ multi-port register files and multiple functional units to support instruction-level parallelism. Fixed (1 word per cycle) bandwidth between cache and register-file might limit processor's ability in spatial/temporal utilization. This paper presents an experimental study of conventional super-scalar processor architecture to determine benefits that we can expect to achieve by enabling variable data bandwidth between the L1 data cache and the register file. Our results demonstrate that by changing the bus width to 64, 128 and 256 bits we can reduce data traffic between the 32KB register-file and 32KB cache up to 29%, 45% and 53%, respectively, while lowering the program execution time by 8%, 13% and 17% on average in comparison to conventional single-word cache access. An adaptive bandwidth cache capable of adjusting the cache bandwidth to workload variation is also proposed.

1 Introduction

1.1 Motivation

With the higher levels of processor performance and widespread of memory latency tolerance techniques, memory bandwidth emerges as a major performance bottleneck. To avoid long delays associated with memory accesses, modern high-performance processors rely on caches. Conventional caches usually store multi-word data blocks per line while delivering only a single word per access. To select a target n-bit word among a number of candidate words, they employ a multiplexor, as shown in 1. When processor contains a single ALU and a 2-read 1-write port register file, such cache organization is reasonable, since no more than one data word is usually needed per cycle. The disparity between the size of data block stored within cache and the size of data delivered to register file is not only invisible but necessary in order to reduce the cost of bus and the I/O pins. However, when processors contain 8 functional units and an 8 read 4 write port register file (e.g. Alpha21264), the fixed (one word) cache bandwidth might lead to performance loss, especially in applications where large Instruction-Level Parallelism (ILP) is possible [1]. For example, Burger et al [2] report between 11% and 31% of the total memory stalls observed in several SPEC benchmarks

P.-C. Yew and J. Xue (Eds.): ACSAC 2004, LNCS 3189, pp. 212–225, 2004.

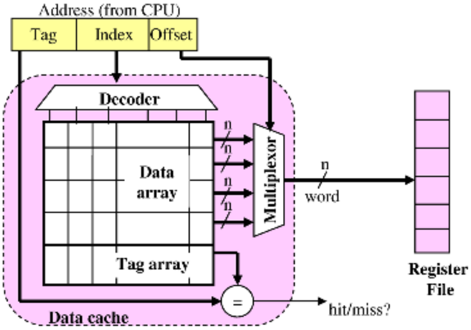


Fig. 1. Cache organization

are due to insufficient memory bandwidth. Rather than rely solely on wider and faster caches and register files, an alternative is to use existing memory bandwidth more efficiently. The main purpose of this paper is to investigate how the cache/register-file bandwidth affects the performance of conventional superscalar processor across SPEC95 benchmarks and to propose cache architecture capable to adapt the bandwidth to workload variation.

1.2 Related Research

A lot of work has been dedicated in the past to analysis of processor-memory interface. McCalpin [3] used the STREAM benchmark to demonstrate that processors had become increasingly unbalanced because of the limited memory bandwidth. Burger, et al [2] simulated on SPEC benchmarks the data traffic dependency on cache block size, associativity, replacement policy, write policy, etc. and proposed three policies (dual-size fetching, subblock prefetching and bus prioritization) to improve L1-L2 memory system interface. Huang and Shen [4] studied what they called intrinsic bandwidth requirement by directly measuring the reuse of values. Ding and Kennedy [5]evaluated the demand and supply of data bandwidth of several scientific kernels through a performance model called balance, and demonstrated the serious performance constraint due to the lack of memory bandwidth. To reduce the memory transfers they advocated compiler-based program transformation such as reuse-based computation fusion and write-back reduction. Johnson, et al, [6] presented a framework for automatic control of cache management techniques capable of determining data placement in run-time. The need to use super-words (i.e. data objects with the size larger than machine words) in order to improve the performance of media programs has been studied in [7].

Several MIPS processor [8] utilizes multiple cache line sizes which are configurable at boot time to minimize the cache miss rate. Virtual Cache Lines (VCL) [9] supports a fixed cache block size for normal references, and fetches multiple sequential cache blocks when the compiler detects high spatial reuse. The line size is sent to the processor at run time by special instructions. The stride prefetching cache [10] also utilizes specific hardware to change the line size based on profiling or other compiler transformations. In [11] is proposed to vary line size based on spatial locality of the miss fetched data, which is detected from the access pattern to the cache by the compiler. The selective sub-blocking technique relies either on hardware [12], [13] or software [14] predictors to track the portions of cache blocks that are referenced by the processor. On a cache miss, the predictors are consulted and only previously referenced (and possibly discontinuous) portions are fetched into the cache, thus conserving memory bandwidth. A sectored cache [15]is proposed to access variable-sized fine-grained data through the annotated memory instructions. The dual data cache [16] selects between two caches, also in hardware, tuned for either spatial locality or temporal locality. These techniques employ line-size selection algorithms that are designed for affine array references and are thus targeted to numeric codes. Run-time adaptive cache management scheme is presented in [6].

1.3 Contribution

This paper presents an experimental study of data cache accesses in conventional super-scalar architecture to determine benefits that we can expect to achieve by enabling variable data bandwidth between the L1 data cache and the register file. Similarly to [1], we analyze performance of modern super-scalar processor using a set of SPEC95 benchmarks. In contrast to previous studies which investigated bandwidth variation between cache and main-memory, we centralize here on bandwidth between data cache and register file. Moreover, unlike [14, 15], we neither employ specific compiler transformations nor restrict ourselves to specific multimedia applications. The study report on unexploited bandwidth potential, observed on SPEC95 benchmarks, and proposes new cache architecture capable of adapting the cache-register bandwidth to workload variation.

The paper is organized as follows. Section 2 presents our simulation methodology. Section 3 shows the results. Section 4 outlines the proposed adaptive bandwidth cache. Section 5 concludes the paper.

2 Methodology

2.1 Simulation Environment

We used SimpleScalarfs [10] sim-outorder to collect our results. SimpleScalar provides a simulation environment for modern out-of-order processors with speculative execution. The simulated processor contains a unified active instruction list, issue queue, and rename register file in one unit called the reservation update unit (RUU). The RUU is similar to the Metaflow DRIS (deferred-scheduling, register-renaming instruction shelf) [11] and the HP PA-8000 IRB (instruction reorder buffer). Separate banks of 32 integer and floating point registers make up the architected register file and are only written on commit. Table 1 summarizes the important features of the simulated processor. The baseline configuration parameters are roughly those of a modern out-of-order processor.

Table 1. Baseline configuration of simulated processor

Parameter	Value
RUU size	80 instructions
Load/store queue size	40
Fetch queue size	8 instructions
Fetch width	4 instructions/cycle
Decode width	4 instructions/cycle
Issue width	4 instructions/cycle (out-of-order)
Commit width	4 instructions/cycle (in-order)
Functional units	4 Integer ALUs (arithmetic, logical, shift, memory, branch ops), 1 integer multiply/divide
Branch Prediction	Combining: 4K 2-bit selector, 12-bit history; 1K 3-bit local predictor, 10-bit history, 4K 2-bit global predictor, 12-bit history
Register file (2 banks)	32 int. registers, 32 fp. registers
Status	6 registers
L1 data-cache	16K, s-way (LRU), 32B blocks,
	1 cycle latency
L1 instruction-cache	16K, 2-way (LRU), 32B blocks,
	1 cycle latency
L2 cache	Not available
Memory	18 cycles

In our simulation we assumed that:

1. The size of the register file is sufficient for processing the programs;
2. The number of accessing ports in RF is sufficient to run the program without port-sharing hazards;
3. Simultaneous register file accesses can be executed in parallel;
4. The data located in the same cache block can be loaded from the cache in parallel.

We experimented with four sizes of processor-cache bus (32 bit, 64 bit, 128bit, 256bit), and four values of data cache associativity (s), namely, s=1 (direct map cache), and s=2, 4, 8 (set-associative cache).

2.2 Benchmarks

A goal of this study is to investigate the impact of cache-RF bandwidth on processor performance even in applications outside the multimedia domain. We experimented with seven typical SPEC95 benchmark programs representative for both integer and floating point applications [13]. The training input sets (*train*) were selected for the benchmarks because they did not take too long to simulate to completion while maintaining behavior to simulating the reference

Table 2. Benchmarks and descriptions

Benchmark	Description	Symbol	Inst.($\times 10^6$)	%mem	%loads
Integer benchmarks (CINT95)					
129.compress	Compresses large text files (16MB) using adaptive Limpel-Ziv coding	comp95	35.7	37.4	64.9
130.li	Lisp interpreter	li	183.3	42.5	62.9
132.ijpeg	Performs jpeg image compression with various parameters.	ijpeg	1462.5	25.5	69.2
147.vortex	Builds and manipulates three interrelated databases.	vortex	2520.2	52.6	58.6
Floating point benchmarks (CFP95)					
102.swim	Solves shallow water equations using finite difference approximations.	swim	849,9	31.0	77.8
107.mgrid	Calculation of a 3D potential field in a 513x513 grid	mgrid	14292,3	36.9	96.2
110.applu	Solves matrix system with pivoting.	applu	531.9	25.5	81.5

data sets in full. Since our focus was on the processor-cache interface, we chose the workloads that sustain high miss rates. Table 2 characterizes the benchmarks in terms of the total amount of instructions executed, the percentage of memory instructions and the ratio of load operations among the total memory operations. Table 3 lists the average miss ratio obtained for 32KB cache (32B block size) by using the sim-cheetah simulator [2]. Each benchmark was run to completion.

2.3 Evaluation Approach

Our goal was to simulate the impact of cache-RF bandwidth on the processor performance. We modified the cache.c and sim-outorder programs of the SimpleScalar simulator [12] to dynamically count the data-transfers between register-file and L1-data cache as well as bus utilization while varying the bit-sizes of bus, which connects the register file and cache. The data bandwidth was measured as the number of machine words which can be transferred in parallel between the register-file and cache due to both load and store operations. For the given bus limit of 256 bits, we used eight counters, which enumerated data transfers of 1, 2, 3c 8 machine words in size, respectively. We assumed that the data size of the first reference R[0] to cache block was always one word wide. Whenever the next cache reference R[j] was encountered, we checked whereas it was of the same type (e.g. load) and to the same cache block as the previous reference, R[i-1]. If all these conditions held, we computed the word-distance (d) between R[i] and the first reference R[0] to the block, and incremented the counter which counted words of size d, while decrementing the counter which counted words of size d-1. Thus, at the end of the program run, the counters indicated the data traffic distribution across the eight bus sizes. The final results displayed also the execution time (measured through the number of total clock cycles) taken by the benchmark; the number of data cache accesses and the bandwidth utilization

Table 3. Miss ratio of 32KB data cache (block size 32B)

Bench- mark	Miss ratio (%)			
	s=1	s=2	s=4	s=8
comp95	5.65	5.47	5.32	5.315
li	0.46	0.23	0.21	0.19
ijpeg	0.83	0.34	0.27	0.25
vortex	1.46	1.01	0.84	0.73
swim	6.65	3.98	3.39	3.29
mgrid	2.44	1.46	0.97	0.98
applu	1.57	1.23	1.22	1.21

ratio (Rj) computed for each benchmark as $R_j = N_i/N_{total} * 100\%$, where N_j is the number of data transfers of size j and N_{total} is the total number of transfers.

3 Results

Figures 2-3 list the simulation results in comparison to the results obtained for conventional 32 bit register-cache communication. We observe that enlarging the bit-width between the L1 data cache and the register file allows us to reduce both the number of cache accesses and, the number of total clock cycles required by the benchmarks. The reduction ratio strongly depends on the benchmarks and the allowable bit-width of cache accesses. The number of cache accesses which can be eliminated by adopting 64bit, 128bit and 256bit wide buses can vary from 31%, 47%, and 53%, respectively, for vortex, down to 8% for *mgrid*. Also, the reduction in clock cycles can be as much as 13%, 18%, 19% (see li benchmark) at 64bit, 128bit and 256bit bus size, respectively, and as low as 4%-6% (for

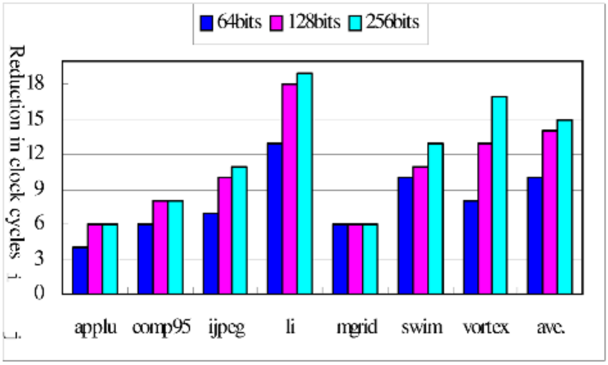


Fig. 2. Reduction in clock cycles in comparison to fixed 32 bit cache bandwidth (direct-map data cache)

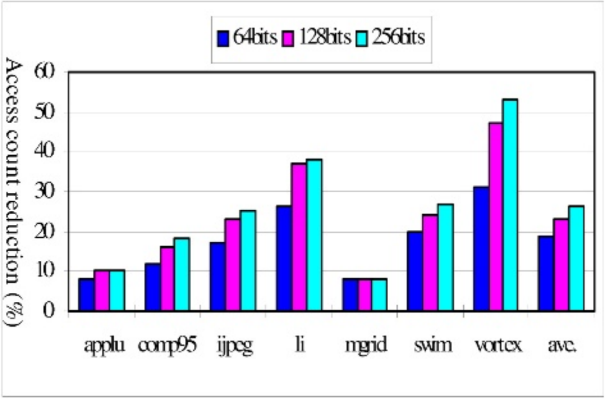


Fig. 3. Reduction in cache accesses (direct-map data cache)

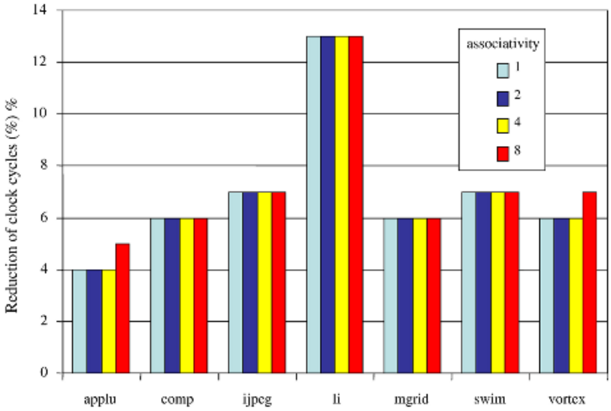


Fig. 4. Impact of cache associativity

applu). The right-most columns (*ave.*) in the figures show the average values across the benchmark. As these results indicate, restricting the cache-register file communication path to 32bits may slower the processor's performance by 10-15% on average, while increasing the cache access count by 17-26%.

To investigate the influence of cache configuration on the results, we repeated simulations by changing associativity of L1 data cache in the baseline processor from 1 to 8. Figure 4 depicts the results for the 64 bit wide bus. We see that the total number of clock cycles required by each program does not depend much on cache associativity. The same trend has been observed for the number of cache accesses as well.

Figure 5 shows the bus utilization statistics. The 32-bit wide cache accesses are the most frequent; they consume from 55% up to 91% of the total data traffic through the bus. When the bus becomes wider, the 64-bit data transfers are replaced by larger data bundles. However, we should note, that accesses

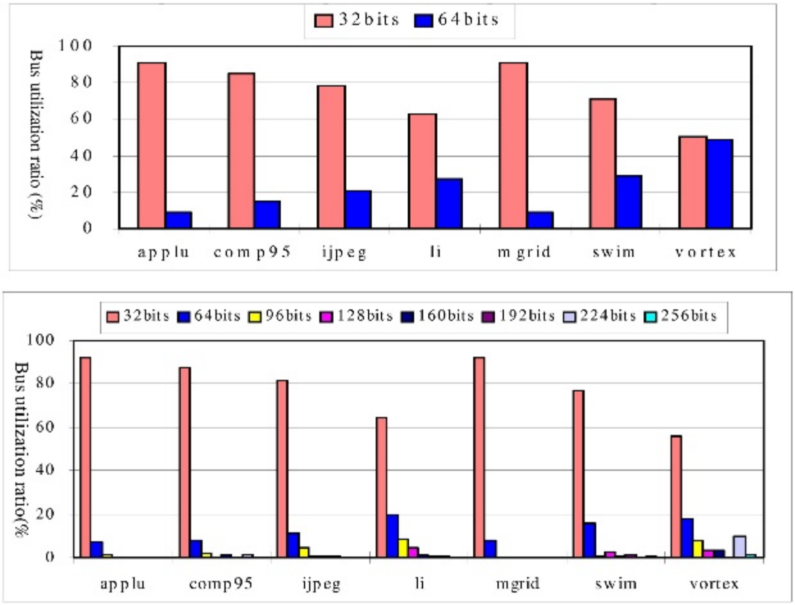


Fig. 5. Bit utilization for 64 bit wide bus (top) and 256bit wide bus (bottom)

wider than 8 bytes in size occur quite rarely (less than 10% of time) in all the benchmarks, but vortex.

To summarize, the results presented in this section have three notable implications:

1. The fixed 32-bit data width between the L1 data cache and register file increases the number of cache accesses by 20-26% on average, slowing the processor by 10-15% in comparison to multi-word transfers even for non-multimedia applications. The required data cache-register file bandwidth is thus significantly larger than that one provided by conventional caches.
2. The cache-register file bandwidth is application-dependent. Fixing the bus size at either extreme (32 or 256 bits) leads either to a poor performance or large area overhead, since multi-byte data transfer occur quite rarely.
3. For almost all the benchmarks, the dual-word data access per clock cycle is sufficient for more than 80% of total cache references. This feature presents an opportunity: if the cache and the register-file could support the dual word traffic, the number of accesses could be reduced by 1/5 on average, thus improving the performance.

These implications lead us to three requirements for on-chip L1-caches. (1) The cache-register file bandwidth should be larger than that of traditional caches, (2) the cache should be managed in such a way as to provide good performance across the entire range of applications, and (3) it should use intelligent (or adaptive) fetching, adjusting its bit-width to application workload without a large

Table 4. A table of instruction set extensions for manipulating AB-cache

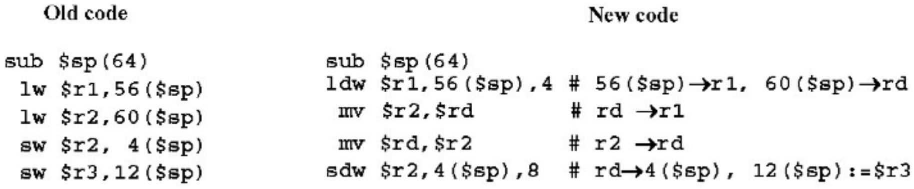
Instruction	Explanation
`ldw rt, off0(rs), off1`	A dual-word load. Access mode, $M = 1$. Data from the cache block pointed to by the block offset bits of $rs + off0$, and by $off1$ is transferred to registers *rt, dr*, respectively.
`sdw rt, off0(rs), off1`	A dual word store. Access mode, $M = 1$. Contents of *rt* and *dr* are stored into the cache block at locations specified by the block offset bits of *rs+off0*, and by $off1$, respectively.

overhead in area or power. As a solution we propose an adaptive bandwidth cache, described in the next Section.

4 Adaptive Bandwidth Cache

4.1 Main Features

The key idea behind our adaptive bandwidth cache (or AB-cache) is to adjust the size of data transferred from (or to) the cache based on spatial locality of memory accesses. When programs have large spatial locality, the cache provides an extended bandwidth, enabling two adjacent words to be accessed in parallel. On the other hand, when spatial locality is small, it reduces the bandwidth down to a single word per access. The AB-cache is software-centric; it relies on the compiler to statically identify the spatial locality of adjacent memory references and replace the next reference to the same cache block by a register transfer operation. (The approach is similar to cache pre-fetching with the difference that it pre-fetches the register-file not cache. Our approach lets software tell the hardware when to perform dual-word fetch, so instead of accessing the same cache block again, the hardware fetches the data directly from the register-file. We augment the processor state with a special registers, $dr, off1$, which are set by software to temporally keep information about destination register and offset of the adjacent reference, respectively. Note, that software is only made aware of the length of the cache block, but not the total cache capacity or associativity. Table 4 outlines the instruction extensions for using the adaptive cache. (Only word accesses are shown, but half-word accesses are handled analogously). Software places values in the *dr* as an optional side effect of performing a load or store. A dual-word load or store specifies a full effective virtual address in addition to off1 number. Figure 6 shows an example code and its transformation. As we see the second and third memory references have been replaced by register transfer operation, *mv*. Note that no additional instructions where added and the performance is identical.



```
    Old code                                        New code

sub $sp(64)                      sub $sp(64)
 lw $r1,56($sp)                  ldw $r1,56($sp),4  # 56($sp)→r1, 60($sp)→rd
 lw $r2,60($sp)                   mv $r2,$rd        # rd →r1
 sw $r2, 4($sp)                   mv $rd,$r2        # r2 →rd
 sw $r3,12($sp)                  sdw $r2,4($sp),8   # rd→4($sp), 12($sp):=$r3
```

Fig. 6. An example function entry code transformed for dual-word data transfers

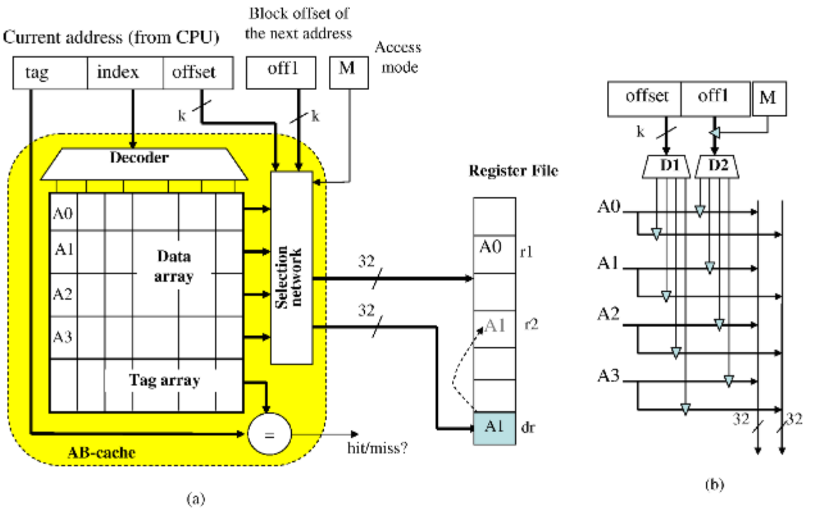


Fig. 7. (a) Block-diagram of AB-cache; (b) internal structure of selection network

4.2 Hardware Architecture

Figure 7(a) depicts a block-diagram of the proposed AB-cache. Additionally to the target address, it receives from CPU the block offset of the next reference ($off1$) and the access mode indicator bit (M), both set by software. (Note that a conventional memory reference has M=0). The k-bit values of the offset and off1 determine which words among the 2K candidates in the cache block are accessed in parallel, while the access mode bit (M) indicates whether the cache operates in a conventional (single word) mode or in a dual-word mode. For example, if $offset$=0, $off1$=1, then the second instruction (ldw) of the transformed code in Fig.4,b will force the AB-cache to read two words ($A0, A1$) in parallel and write them to registers $r1$ and rd, respectively. Figure 7 (b) shows in details the selection network structure, where triangles depict three-state buffers, $D1, D2$ denote decodes. In conventional access mode, the network disables unselective lines from the bus to save power.

4.3 Compiler Support

Determining whether a reference exhibits temporal or spatial locality, and whether this locality is worth being exploited is a well documented research topic [18]-[20]. Similarly to [19], we annotate the references as spatial, if the coefficient of the innermost loop in the reference subscript is smaller than 4 (the cache block size usually considered, i.e.16 bytes corresponding to 4 single precision data word). If the coefficient is parameter, the reference is not tagged spatial. Finding temporal dependences such as self-dependence (x[i],c,x[i]) or a uniformly generated temporal group-dependence (e.g. b(j,i), b(j,i+1)) amounts to simple subscript analysis. These two types of dependences account for a significant fraction of the dependences, as already mentioned in [20].

We use a two step approach to find adjacent cache references which can be performed in parallel. First, we look for two reference, one of which dominates the other, so all paths that cause the subordinate access to be executed cause the dominant to reference to be executed first. Second, we try to prove that the two references always point to the same cache line. In this case, the second reference can be read in parallel with the first. To determine whether two references are to the same cache block, we employ alignment and distance analysis. The former determines the address alignment of each memory reference relative to a cache boundary. The latter detects the byte distance of two static memory references. A load instruction is considered aligned when its cache alignment is the same for each dynamic execution of the instruction. If the difference between two static memory references (address calculations) is constant, then we know the distance between the references. In the initial compiler passes, when array indexes are represented at a high level, we tag them with their source array to aid in distance analysis. We use this tag once the array access has been decomposed into pointer manipulation. For accesses of the form x[i] and x[i+c], our tagging allows us to compute the distance as c. This pattern occurs very frequently in unrolled loops.

Once we know the distance, we can use the alignment to determine if two references are to the same cache line. We find the alignment of the dominant reference relative to the cache line boundary and then find the distance between the subordinate access and the dominant access. Simple arithmetic indicates if the references are on the same cache line. When the distance is 0, we can ignore the alignment information.

5 Estimation

We developed a prototype compiler to output instrumented C-code and then run the modified Simple-Scalar simulator to estimate benefits of the AB-cache. No extra code transformations have been applied to facilitate the spatial reuse and increase aligned memory operations. We simulated the baseline processor architecture (Tables 1, 2 s=1) and compared the results with the results reported in Section 3 for the direct-map cache. We assumed that dual-cache load and store instructions add 1 clock-cycle to the pipeline. Figure 6 outline the results. Though the proposed cache performs by 5-10% worst in comparison to ideal

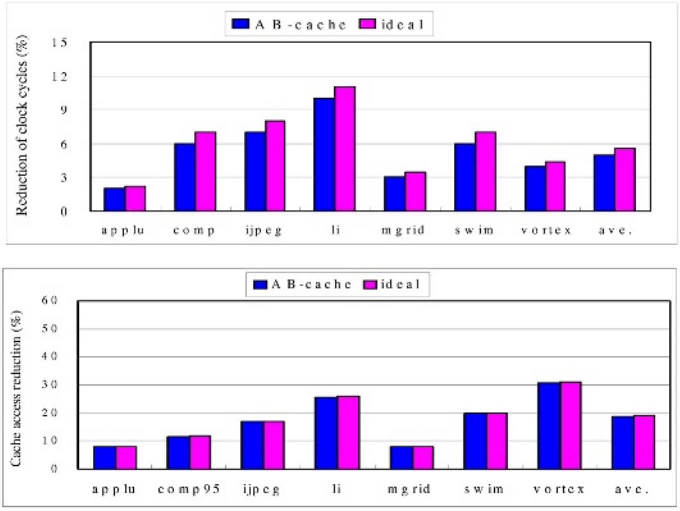


Fig. 8. Normalized execution time (top) and cache access reduction (bottom) in comparison to potential results for 64 bit-wide bus

execution time (i.e. the total number of cycles) presented in Section 2, the benefit of using it is high: in comparison to conventional cache it improves processor performance by 5-15%, lowering the total number of L1 data by a half.

6 Conclusion

This study indicated that fixed 32-bit data width between the L1 data cache and register file increases the number of cache accesses by 20-26% on average, slowing the processor by 10-15% in comparison to multi-word transfers even for non-multimedia applications. The results pointed out that the cache-register file bandwidth should be at least twice larger than currently used one in order to efficiently support even non-memory intensive applications. As a solution, we presented a dual-word cache capable of adaptively adjusting the bandwidth to workload variation. To use the AB-cache we need both the hardware and the compiler modifications. However, it is quite promising. Even our preliminary work had not incorporated program transformations (e.g. [5], [7], which proved to be effective in to facilitating spatial reuse of memory accesses, we could reduce the number of accesses to L1 data cache by half while improving the processor performance by 5-15%.

The mechanism we discussed could be extended to achieve cache bandwidth larger than dual-word. Large improvement is also expected from incorporating code transformations into the modified compiler. Furthermore, with a proper modification of the register-renaming algorithms, we can eliminate many register-transfer operations (currently added to the code) and reduce the number of data transfers at least twice. Moreover, the existing register files already support multi-word reads and writes so moving the data to and from *dr* register looks redundant. Future work also will be dedicated to studying effects of AB-cache on power consumption.

References

1. H.Liao, A.Wolfe, Available Parallelism in video applications, *Proc. Micro-97*, pp.321-329.
2. D.Burger, J.R.Goodman, and A.Kagi, Memory Bandwidth limitations of Future Microrocessors, *Proc. Annual 24th Int. Symp. On Computer Acrhitecture*, pp.78-89, 1996.
3. J.McCalpin, Sustainable memory bandwidth in current high-performance computers. http://reality.sgi.com/mccalpin_asd/papers/bandwidth.ps, 1995.
4. S.A.Huang and J.P.Chen, The intrinsic bandwidth requirements of ordinary programs. *Proc. 7th Int.Conf. on Arch.Support for Programming Languages and Operating Systems*, 1996
5. Ding and K.Kennedy, Memory Bandwidth Bottleneck and Its Amelioration by a Compiler, *Proc. Int.Parrallel and Distributed Process. Symp.*, 2000.
6. T.L.Johnson and W.W.Hwu, Run-time adaptive cache hierarchy management via reference analysis, *Proc. Annual 24th Int. Symp. On Computer Acrhitecture*, June 1997.
7. S.Larsen, S.Amarasinghe, Exploiting Superword Level Parallelism with Multimedia Instruction Sets, P̂roc. ACM SIGPLAN Conf.on Progr.Language Design and Implementation, 2000, pp.145-156.
8. MIPS Corporation, MIPS R3000 hardware manual, MIPS Corporation.
9. K.Inoue, K.Kai, and K.Murakami, High bandwidth, variable line-cache architecture for merged DRAM/logic LSIs, *IEICE Transactions on Electronics*, E81-C(9), pp.1438-1447, Sep.1999.
10. T.-F.Chen and J.-L.Baer, Reducing memory latency via non-blocking and prefetching caches, *Tech.Rep. 92-06-03, Dept. Computer Science and Engineering, Univ.Washington*, Seattle, WA, June 1992.
11. A.Veidenbaum, W.Tang, R.Gupta, A.Nicolau, and X.Ji, Adapting cache line size to application behavior, *Proc. Int. Conf. on Supercomputing*, pp.145-154, 1999.
12. D.Burger, Hardware Techniques to Improve the Performance of the Processor/Memory Interface, *Tech. Rep. Computer Science Dept., University of Wisconsin-Madison*, Dec. 1998.
13. S.Kumar and C.Wilkerson, Exploiting Spatial Locality in Data Caches using Spatial Footprints, *Proc. 25th Annual Int. Symp. On Computer Acrhitecture*, pp. 357-368, June 1998.
14. D.Agarwal, and D.Yeung, Exploiting Application-Level Information to reduce memory bandwidth consumption, *Technical Report UMIACS-TR-2002, Univ.of Maryland, Inst. For Advanced Computer Studies*, 2002.
15. A.R.Lebeck, D.Raymond, C-L.Yang, M.S.Thottethodi, *Annotated Memory References: A Mechanism for Informed Cache Management*, 1999
16. A.Gonzales, A.Aliagas, and M.Valero, A data cache with multiple caching strategies tuned to different types of locality, *Proc. 1995 Int.Conf. on Supercomputing*, pp.338-347, July 1995.
17. V. Popescu, M. Schultz, J. Spracklen, G. Gibson, B. Lightner, and D. Isaman. The Metaflow architecture. *IEEE Micro*, pp.10-13, 63-73, June 1991

18. J.Ferrante, V.Sarkar, W.Trash. On Estimating and Enhancing Cache Effectiveness, *Proc.4th Workshop on Languages and Compilers for Parallel Computing*, 1991.
19. O.Temmam and N.Drach, Software Assistance for Data Caches, *Proc. IEEE HPCA*, 1995.
20. M.S.Lam, et al., The SUIF compiler System, 1992-2001. http://www-suif.stanford.edu

Exploiting Free Execution Slots on EPIC Processors for Efficient and Accurate Runtime Profiling

Youfeng Wu and Yong-Fong Lee

Corporate Technology Group (CTG), Software and Solutions Group (SSG)
Intel Corporation
2200 Mission College Blvd
Santa Clara, CA 95054
{youfeng.wu, yong-fong.lee}@intel.com

Abstract. Dynamic optimization relies on runtime profile information to improve the performance of program execution. Traditional profiling techniques incur significant overhead and are not suitable for dynamic optimization. In this paper, we propose a new profiling technique that incorporates the strength of both software and hardware to achieve near-zero overhead profiling. The compiler passes profiling requests as a few bits of information in branch instructions to the hardware, and the hardware uses the free execution slots available in a user program to execute profiling operations. We have implemented the compiler instrumentation of this technique using an Itanium research compiler. Our result shows that the accurate block profiling incurs very little overhead to the user program in terms of the program scheduling cycles. For example, the average overhead is 0.6% for the SPECint95 benchmarks. The hardware support required for the new profiling is practical. We believe this will enable many profile-driven dynamic optimizations for EPIC processors such as the Itanium processors.

1 Introduction

For EPIC processors like the Itanium ([3][13][18]), the compiler needs a certain amount of knowledge about a user program to generate more efficient code. A static compiler obtains this information from profiling the program with its training data. A growing interest has been moving toward profiling and optimization at runtime with actual input data. This "runtime profiling and optimization" environment requires the collection of program profiles at runtime efficiently. The most commonly used profiles are block/edge and path profiles. The best-known edge and path profiling algorithms are from [4] and [5]. However, the instrumented code in the user program typically incurs about 30% overhead with block/edge profiling and 40% with path profiling. Our experience with Itanium processors shows that the overhead of block profiling ranges from 14% to 42% for SPECint95 benchmarks. In the context of dynamic optimization, this overhead is not acceptable during the execution of a deployed user program.

Recent researches have focused on sampling based profiling to reduce the profiling overhead ([1][19]). Sampling based profiling could potentially lose profile accuracy.

P.-C. Yew and J. Xue (Eds.): ACSAC 2004, LNCS 3189, pp. 226–240, 2004.

It also requires operating system supports and incurs noticeable runtime overhead due to software interrupts. We are interested in a method such that accurate profiles can be obtained without OS support and without noticeable runtime overhead.

Here we present an approach that combines both hardware and software support to greatly reduce profiling overhead. We first focus on *block profiling*, namely to find the execution frequencies of the basic blocks in a user program, and later extend the technique to edge profiling. To collect block profile information, the traditional profiling technique inserts a load, an increment, and a store to each block that needs profiling. In our new profiling technique, the compiler passes profiling requests as a few bits in branch instructions to the hardware, and the hardware uses the free execution slots available during the execution of the user program to collect profile information. With this technique, the program execution with profiling can run almost as fast as without profiling. This will enable many profile-driven dynamic optimizations for EPIC processors such as the Itanium processors.

As an example, we look at the control flow graph (CFG) in Fig. 1. We only need to profile blocks b, c, d, f, and h, and the frequencies for other blocks can be derived from those of the profiled blocks. The traditional profiling technique would insert three instructions in each of the profiled blocks to load a counter from memory, increment the counter, and store the counter to memory. The inserted profiling code is listed in Fig. 1(a). For our new technique, we first assign ID's to the blocks that need to be profiled. The ID's are in the range of [1, …, number of profiled blocks]. If a block already has a branch instruction and the profile ID fits into the ID field, we encode its ID in the field. The modified branch instructions are listed in Fig. 1(b). At runtime, the hardware derives the memory address of a profile counter from the ID field in a branch instruction and automatically generates load/increment/store instructions. These hardware-generated profile update operations are executed in free execution slots available in the user program.

Our proposed technique achieves "near zero" profiling overhead by combining two collaborative techniques: (1) powerful compiler analysis to insert minimal profiling instructions, and (2) hardware that asynchronously executes profile update operations in free execution slots available during the user program execution. The following are several interesting issues solved in our approach.

We need an algorithm to select a minimal number of blocks to profile so that the execution frequencies for other blocks can be derived from them. Furthermore, we require as many selected blocks to contain branch instructions as possible. In the following we will call a block selected for profiling a *profiled block*, which is further classified as a *branch block* or a *non-branch block* depending on whether it contains a branch instruction.

Although most profiled blocks have branch instructions in them, some may contain no branch instructions to encode the profile ID's. We provide an explicit instruction (prof_id) to pass profile ID's to the hardware for non-branch profiled blocks. Our experiment shows that we only need to use the prof_id instructions in very few blocks. We will use the term *profiling instructions* to refer to the instructions inserted by the compiler for passing profiling needs to the hardware. They include the *prof_id* instruction as well as the *initprof* and *setoffset* instructions that will be described later. The hardware will translate the information carried by profiling instructions into *profile update operations* to manipulate profile counters.

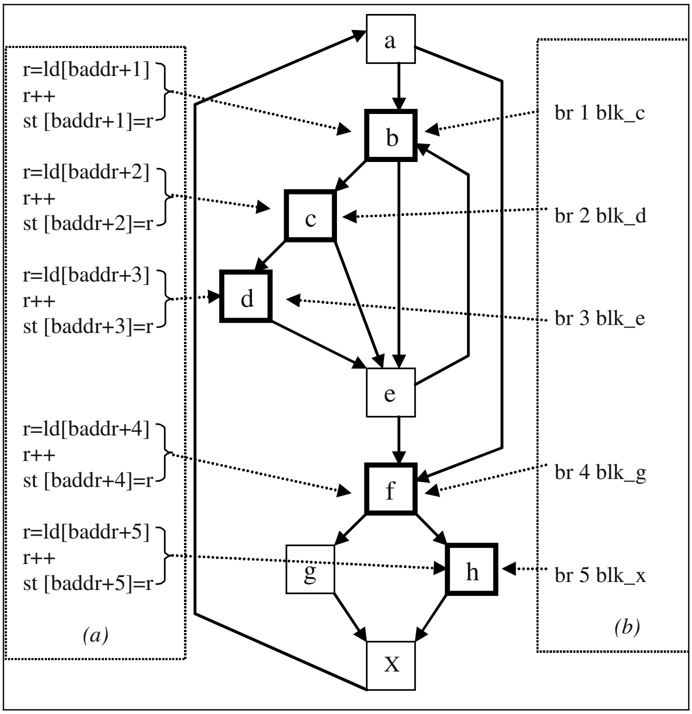


Fig. 1. An illustration example

A branch instruction may allow only for a few bits of profile ID. The number of profiled blocks in a function may be more than that the ID field can represent. We apply an algorithm to partition a CFG into regions such that the ID field can adequately represent the total number of ID's in each region. We also pass additional information in the region entry blocks to assist the hardware to derive profile counter addresses from these ID's.

The hardware intercepts profiling requests and generates instructions to update profile counters. Although the *profile update operations* are inserted into free execution slots available in a user program at runtime, they may have longer latencies than instructions from the user program. We do not want their execution to delay the user program. To achieve this, we devise a method for the hardware to arrange the update operations so that they do not stall the execution of the user program.

The rest of the paper is organized as follows. Section 2 describes relevant background material about Itanium. Section 3 discusses the architectural support. Section 4 presents the details of our new profiling technique, including the compiler algorithms for block selection and graph partitioning, and the hardware support. Section 5 gives our experimental results. Section 6 extends our technique to do edge frequency profiling. Section 7 describes related work. Section 8 gives concluding remarks and future work.

2 Relevant Itanium Features

Itanium [13] provides many features to aid the compiler in enhancing and exploiting instruction-level parallelism (ILP). These include an explicitly parallel instruction set (EPIC), large register files, and architectural support for predication, speculation, and software pipelining. An Itanium program must explicitly specify *instruction groups*, each of which is a sequence of instructions that have no register flow and output dependencies. Instruction groups are delimited by *architectural stops* in the code. Because an instruction group has no register flow and output dependencies, its instructions can be issued in parallel without hardware checks for register dependencies.

The compiler divides an instruction group into *bundles*. A bundle contains three instruction slots as shown below. Each instruction slot takes 41 bits. Most instructions take one instruction slot, while those taking a link-time symbol as an operand may take two slots. The template field specifies the mapping of instruction slots to execution unit types.

slot1	slot2	slot3	template

We have observed a significant number of free execution slots available in many machine cycles that can be utilized for profiling. The free execution slots arise from the following causes:

- Not all the possible mappings of instructions into functional units are permitted. For example, a bundle can contain at most two load operations. Thus an instruction group with three load instructions must be assigned into two bundles. This would leave three free execution slots in the two bundles if there were no other instructions in the group.
- When the number of instructions in an instruction group is not a multiple of the bundle size, the last bundle for the instruction group will have free execution slots due to fragmentation.
- The ILP available in applications may not be sufficient to fully utilize the machine width all the time. This is especially the case for control-intensive scalar code. Incidentally, an earlier study conducted by Diep et al. [9] reports an IPC of 1.05–1.25 for four integer benchmarks and 1.0–1.9 for three floating-point benchmarks on a 4-way PowerPC 620 processor. In [8] it is reported that on large commercial applications, average cycles-per-instruction (CPI) values may be as high as 2.5 or 3. With 4-way instruction issue, a CPI of 3 means that only one issue slot in every 12 is being put to good use.

To reduce code size, Itanium processors allow for in-bundle stops to pack dependent instructions into the same bundle to reduce explicit no-ops. The hardware dynamically inserts no-ops when the bundle is expanded before execution. We will modify the no-op insertion logic to insert profiling operations when no-ops are needed.

Another relevant Itanium architectural feature is the "multi-way branches", in which multiple branch instructions are placed in the same bundle and executed in the same cycle. Multi-way branches are helpful in reducing control dependence height and should be used as much as possible to increase ILP. Our partitioning algorithm will preserve multi-way branches.

3 Profiling Instructions and Registers

In this section, we outline extensions to the Itanium architecture required by our profiling technique. We assume that the processor has a special 64-bit status register dedicated for profiling. We call it the *profile information register (pir)*. This register contains the following fields with their respective widths in bit:

base_address 40	unused 3	offset 20	flag 1

In addition to an extension to the branch instruction to use the 8-bit branch hints in the current Itanium branch instructions for profile ID's, we also need several new instructions given in the following table.

Instructions	Descriptions
prof_id ID	Pass the profile ID to the hardware.
initprof baseAddr	Initialize pir.base_address as baseAddr.
setoffset offset	Set pir.offset as offset.
startprof	Set pir.flag as 1 to activate or resume profiling.
stopprof	Reset pir.flag as 0 to deactivate profiling.

By default, the value of the ID field in a branch instruction is zero. A non-zero ID is set for a profiled block. We assume that the ID field uses K bits, where K >=1 and is assumed to be K = 8 in this study. For a non-branch block, we use the instruction "prof_id ID" to explicitly pass a profile ID to the hardware.

The pir register is preserved on function calls. On entering a function, the value of pir except for its flag is initialized as zero. We require that each profiled function execute the "*initprof baseAddr*" instruction in its entry block to initialize pir.base_address. Normally, the "*startprof*" instruction is executed in the main function of a program to activate profiling. The dynamic optimizer can execute the "*stopprof*" and "*startprof*"instruction to stop and resume profiling.

We also need a few scratch profile registers, which will be described later. Note that none of the profiling instructions will affect the correctness of the user program. They can be safely ignored by a particular processor implementation if needed.

4 Details of Our Profiling Technique

With the new profiling instructions and registers, our technique performs compiler analysis and instrumentation to place profiling instructions in a user program. The compiler also allocates memory space for recording profile data. When the instrumented program runs, the hardware intercepts the profiling requests and translates them into profile update operations. The update operations are inserted into the free execution slots available during the execution of the user program.

4.1 Compiler Instrumentation

The compiler instrumentation performs the following tasks:

- Selecting profiled blocks
- Partitioning CFG into regions
- Inserting profiling instructions and adding profile ID's to branch instructions in profiled blocks

4.1.1 Selecting Profiled Blocks

The compiler first selects a set of profiled blocks. We extend the Knuth algorithm in [12] to find the minimal set of profiled blocks in a function. The original Knuth algorithm performs the following steps:

- Partition the CFG nodes into equivalence classes. Nodes b1 and b2 are equivalent, denoted by b1 ≡ b2, if there is another block b such that both b->b1 and b->b2 are CFG edges.
- Construct a graph with the equivalence classes as nodes and the original CFG blocks as edges. Select a maximal spanning tree from the graph. The original CFG blocks corresponding to the edges on the spanning tree do not need profiling, and all other blocks are profiled blocks.
- For each block that does not need profiling, find a set of profiled blocks from which the profile information for this block can be derived.

In general, it may be impossible to require that every profiled block contain a branch instruction. We thus extend Knuth's algorithm such that as many profiled blocks have branch instructions in them as possible. This is achieved by modifying the second step of the algorithm. Namely, during the maximal spanning tree computation, we treat blocks containing no branch instructions as having a very large weight. Consequently, it is more likely for these blocks to be included in the maximal spanning tree and thus excluded from profiling.

Assume n branch blocks and m non-branch blocks in a function are selected for profiling. Then the compiler allocates n + m memory locations to store profile information for these blocks.

4.1.2 Partitioning CFG into Regions

The input to the partitioning algorithm is a CFG, and each CFG block is marked as either a profiled block or not, and, for a profiled block, as either a branch or non-branch block. We want to partition the CFG into single-entry regions such that the number of regions is small and each region contains no more than 2^K - 1 branch blocks.

Lee and Ryder have formulated the problem of partitioning an acyclic flow graph into single-entry regions subject to the size constraint [14]. They proposed two approximation algorithms for the problem, which was shown to be NP-hard. Here we are interested in partitioning a cyclic flow graph, and only branch blocks will be counted in the size constraint. Thus, we extend their algorithms by considering the following factors:

- Rather than limiting the number of blocks in a region, we limit the number of branch blocks.
- Cycles are allowed within a region as long as the region has only one single entry block.
- When two or three blocks are grouped together to allow for multi-way branches, we force them into the same region. This is to avoid using a block later in the sequence as a region entry block. Otherwise, the profiling instruction to be inserted in the region entry block will prevent it from being grouped with the earlier branch(es).

The extended algorithms partition the CFG into single-entry regions such that each region contains no more than 2^K-1 *branch blocks*. Note that a region may contain any number of non-branch blocks. Assume t regions are formed. We name the regions as R0, R1,…, Rt-1. For profiling efficiency, we name the region headed by the function entry block as R0. The number of branch blocks in Ri is denoted by NB(Ri) and the number of non-branch blocks in Ri is denoted by NN(Ri). Let size(Ri) = NB(Ri) + NN(Ri).

For each region Ri, the compiler assigns an ID number in the range of [1,.., NB(Ri)] to each of the branch blocks, and an ID number in the range [NB(Ri)+1, size(Ri)] to each of the non-branch blocks. Remember that ID=0 is assigned to branches that do not need profiling. Let ID(b, Ri) be the ID number of a profiled block b in Ri.

Assume the starting address of the profile storage is base_address. For the j'th block bij in region Ri, the starting address of its profile counters is "base_address + ID(bij, Ri) - 1 + $\sum_{v=0}^{i-1} size(R_v)$."

4.1.3 Inserting Profiling Instructions and Modifying Branch Instructions

The compiler inserts an "initprof baseAddr" instruction in the function entry block to load the base address of the profile counters storage into the profile information register.

For each region Ri, 0<i<t, the compiler inserts a "setoffset $\sum_{v=0}^{i-1} size(R_v)$" instruction in its entry block. We do not need to perform this operation for R0 since its offset is zero.

For each profiled block bij in region Ri, if it is a branch block, the compiler modifies the branch instruction of the block from "br target" to "br ID(bij, Ri), target"; otherwise, the compiler inserts an instruction "prof_id ID(bij, Ri)" into the block bij.

A profiled block may have more than one profiling instruction inserted. For example, it is possible that an entry block with an "initprof" instruction also has a "prof_id" instruction inserted. We can combine the two instructions by extending the initprof and setoffet instructions to carry an ID field. In this case, the following peephole optimization can be applied.

- If an "initprof addr" and a "prof_id ID" are in the same block, replace them with "initprof addr, ID".

- If a "setoffset offset" and a "prof_id ID" are in the same block, replace them with "setoffset offset, ID".

With the peephole optimization, a profiled block will have at most one profiling instruction inserted. These peephole optimizations have not been implemented for this paper.

4.2 Profiling Hardware

At runtime, when a branch instruction "br ID, target" or "prof_id ID" retires, the profiling hardware first checks pir.flag. If the flag is set, it then checks the ID value. If the ID value is zero, the hardware does not need to profile the block. Otherwise, the hardware generates the address of the profile counter as

address = pir.base_address + pir.offset + ID - 1

and performs the following update operation.

++(*address)

We want the hardware to perform the update operations asynchronously with the user program so as not to impact critical user program execution. One way is to utilize the free execution slots available in a EPIC processor. In this method, each update operation can be implemented by the following sequence of operations (assume the profiling hardware places the address of a profile counter into register raddr):

```
r = ld [raddr]
r = r + 1
st [raddr] = r
```

These operations are placed into a buffer, and the dispatch unit of the processor inserts the buffered operations into the user program execution stream whenever free execution slots become available. Notice that the update operations do not have to complete within a time limit if no free execution slots are available, as long as they eventually complete. However, the buffer is of a limited size and it may fill up when free execution slots are limited. When that happens, we can discard some of the buffered operations. This will reduce profiling accuracy but may still be acceptable if a majority of the profile update operations are performed. This approach is shown in Fig. 2.

Note that we will need to use an address register and an accumulator register during the three update operations (consisting of a load L, an increment I, and a store S) to temporarily store the address and the loaded value. If we are to interleave the operations from multiple updates, we will need multiple address and accumulator registers for simultaneously executing the update sequences.

To minimize the impact of the update operations on the user program execution, we mark the loads in the update operations to bypass the first level cache (L1) so they won't compete for cache resources with loads in user programs*. Therefore, we need to arrange the corresponding increment and store operations C and C + 1 cycles, respectively, later than the load instruction, where C is the L1 cache miss latency,

* Itanium processor allows a load to carry a hint to indicate no temporal locality at the L1 (nt1), L2 (nt2), or all cache levels (nta).

such as 5 cycles. The update operations can be arranged in a simple pipeline to hide the latencies.

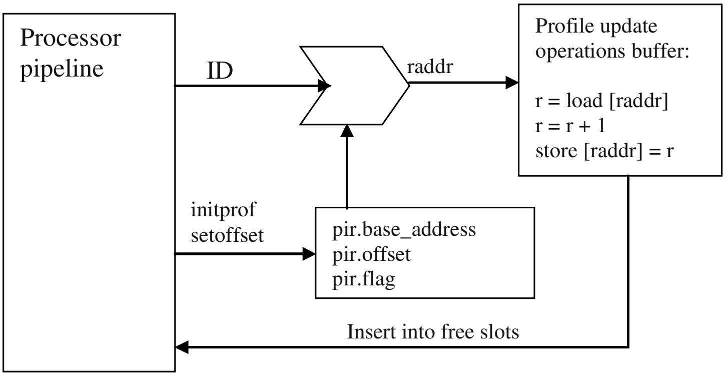


Fig. 2. Profiling hardware

Assume we have C address registers and C accumulator registers that are used in a round-robin fashion. Namely, if the update operations i (consisting of L_i, I_i, and S_i) use register j, then the update operations i+1 (consisting of L_{i+1}, I_{i+1}, and S_{i+1}) will use the register (j+1) modulo C. We will arrange the profile update operations buffer as a circular buffer. Each entry is a three-instruction tuple, consisting of "L_{i+C+1}, I_{i+1}, S_i", where L_{i+C+1} is the load for update i+C+1, I_{i+1} is the increment operation for update i+1, and S_i is the store operation for update i. As long as we insert instructions into free execution slots in the buffered instruction order, and we do not insert more than three instructions in a single execution cycle, the update operations will produce the correct profiling result. This pipelined execution is illustrated in Fig. 3 and it will hide the entire C-cycle latencies of the loads. If we use more address and accumulator registers we can insert more instructions in a single execution cycle.

Although we expect the current 6-wide issue Itanium processor to have enough free execution slots for the profiling update operations, for a narrower processor that do not have enough free execution slots available, we can use dedicated hardware to perform the update operations. The dedicated hardware only needs to perform the update efficiently so it can be easily built. With the dedicated hardware, little or none of the profile update operations will be discarded, and the resulting profile data can be more accurate.

L1	..	..
L2	..	..
L3	I1	..
L4	I2	S1
..	I3	S2
..	I4	S3
....		S4

Fig. 3. Pipelined execution of profile update operations

This approach uses the hardware as shown in Fig. 4 to perform the update operations. After the profile counter ID is used to compute the profile data address addr, the update operation ++(*addr) is sent to the profile update operations buffer and performed by the dedicated hardware. The dedicated hardware performs the update directly without bringing data to the processor and then writing it back.

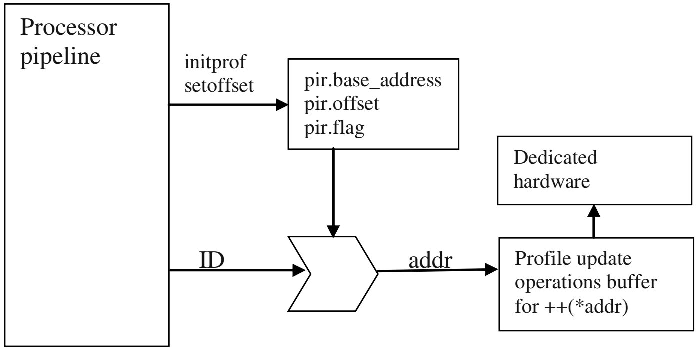


Fig. 4. Dedicated profiling hardware

5 Experimental Results

We have implemented the compiler instrumentation described earlier in an Itanium research compiler. Our experiment used the SPECint95 benchmarks as the test programs.

Table 1 shows the compile-time statistics for our profiling technique. With the extended Knuth algorithm, we need to statically select, on average, about 38% of the blocks for profiling, and only 2.3% of the profiled blocks do not contain branch instructions. The profiled blocks selected account for about 30% of the total *dynamic* block frequency.

By using an 8-bit ID field in branch instructions, each function can be partitioned into 1.15 regions on average. About 3.55% of the blocks are function entry blocks, and 0.11% of the blocks are region entry but not function entry blocks. Overall, about 5.8% of the blocks need to have profiling instructions inserted.

Since the new profiling technique requires hardware support and we don't have a cycle-accurate simulator to evaluate this new technique, we use "schedule length" increase as our metrics for the profiling overhead. For statically scheduled EPIC microprocessors, the schedule length represents the portion of the execution time spent in the CPU core, without considering such microarchitectural stalls as branch miss, cache miss, etc. We believe that the new profiling technique should not noticeably increase the stalls related to branch and cache misses, and therefore the overhead in term of "schedule length" should be a good estimate of the actual overhead in term of overall performance.

In Fig. 5, we compare the overhead of our new profiling technique with that of the traditional block profiling (with Knuth's optimization). Both are compared against the baseline with no profiling. The traditional block profiling incurs about 22%

Table 1. Static statistics for profiling instrumentation

SPECint95	%blk profiled	%blk w/ prof_id	%blk w/ initprof	%blk w/ setoffset
compress	41.4	3.3	4.07	0.00
gcc	39.5	2.2	2.32	0.71
go	44.8	4.4	2.46	0.02
ijpeg	36.4	4.6	5.71	0.00
li	32.0	0.8	6.16	0.00
m88ksim	38.4	1.3	3.79	0.36
perl	39.4	0.8	1.64	4.50
vortex	33.5	1.0	2.24	0.15
Average	38.2	2.3	3.55	0.11

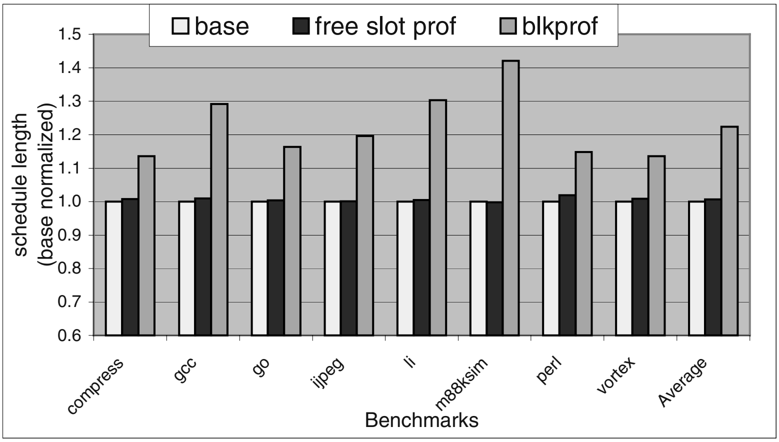


Fig. 5. Comparing the overhead in our technique and traditional block profiling

slowdown on average, ranging from 14% for compress to 42% for m88ksim. By contrast, our method has an average overhead of 0.6% because almost all the initprof, setoffset, and prof_id instructions can be scheduled into free execution slots at compile time, without increasing code size and execution cycles.

6 Extension to Edge Profiling

We may extend our technique for block profiling to do edge profiling. In this case, the profiled edges can be identified with an algorithm proposed in [4]. The CFG partitioning algorithm will be modified to use the number of profiled branch edges as the size constraints. The branch instruction will carry one of the following hints:

- "tk", for profiling the taken edge when the branch takes
- “nt”, for profiling the not-taken edge when the branch fall-through
- "both", for profiling the taken edge when the branch takes and the not-taken edge when the branch fall-through

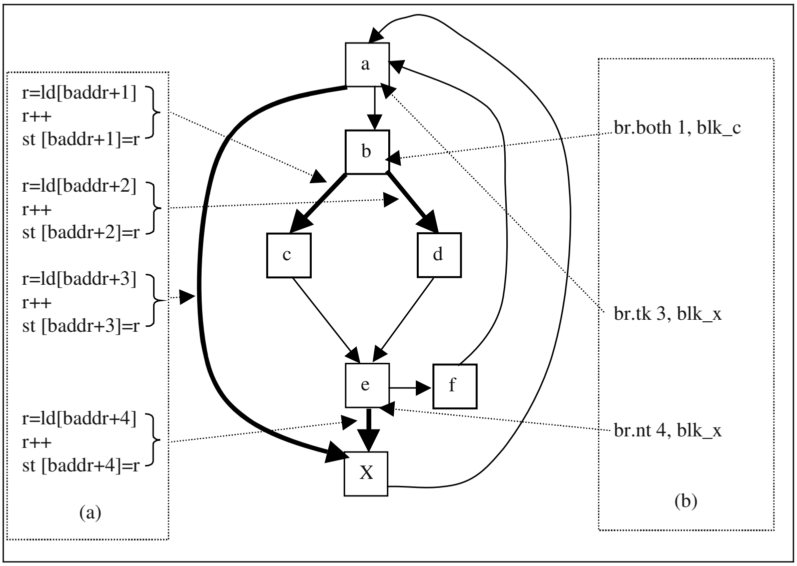


Fig. 6. An example of collaborative profiling

Notice that the "branch.both ID target" instruction may modify two profile counters, one for the taken edge and the other for the fall-through edge. We require that the two edges be assigned two consecutive IDs so the hardware can determine the counter addresses easily.

We use the control flow graph (CFG) in Fig. 6 to describe the edge profiling technique. In this CFG, we only need to profile the edges a•x, b•c, b•d, and e•x, and the frequencies for other edges can be derived from those of the profiled edges. The traditional edge profiling technique would insert three instructions in each of the profiled edges to load a counter from memory, increment the counter, and store the counter back to memory. The inserted profiling code is listed in Fig. 6 (a). Our new technique only needs to modify three branch instructions as shown in Fig. 6 (b). The processor derives the profile operations from the modified branch instructions and executes them without impacting the execution of the user program.

7 Related Work

Software only approaches to profiling can be found in [4][5]. They slow down user programs by about 30% so they may not suitable in the context of dynamic optimization. The main problem with software only approaches is that the inserted instructions compete with the user program for machine resources (e.g. the architectural registers), and they impose dependence that may lengthen the critical paths in the user program.

The sampling based profiling technique in [1][19] collects statistical block profiles by periodic timing interrupts. This technique requires operating system support, and the profile information collected may not be accurate enough for traditional profile-guided optimizations although it is shown to be useful for code layout optimization.

Bursy tracing [2] is a proposed technique to collect profile information via software controlled sampling. It generates two versions of code, one for profiling and one for optimized execution without profiling, and switch from the optimized code to the profiling code infrequently to collect sampling based profile information efficiently. The major drawback is the need to duplicate the code.

In [15][16], a hardware approach is proposed for identifying hot regions and collecting branch profiles for the hot regions. However, this approach requires significant hardware and also needs operating system support. In addition, the profiles collected by using this technique may not be complete and some edges in a hot region may be missed due to cache conflicts and the lack of backup storage. Heuristic patching of the profile may lead to inaccuracy.

Processors, such as Intel Pentium and Itanium, have software readable branch target buffers (BTB). In [6] a technique is described on how to cheaply estimate a program's edge execution frequencies by periodically reading the contents of BTB. In [7] a hardware called a profile buffer is proposed, which counts the number of times a branch is taken and not taken. These techniques require software interrupt to examine the hardware buffers to obtain sampled profiles.

In [10] a profiling technique is implemented for counting the number of times each region exit takes. The dynamic compiler instruments region exit and an 8K entry 8-way set associative hardware array caches counters indexed by the exit point identifiers. This method is specifically targeted for dynamic region formation and expansions.

In [11], the instrumented instructions are scheduled in a software pipelining fashion and that together with speculation and predication reduce the overhead of edge profiling to about 3.3% on an eight-wide machine, without considering cache and branch overhead. However, this technique may generate more memory traffic than necessary, as it may speculatively execute loads and stores for profiling. Furthermore, this technique may still have the instruction cache penalty due to code size expansion. In [17], it is reported that profiling instrumentation increases the text size of a program by a factor of 2-3. On an Itanium processor, since the in-bundle stops can be used to reduce explicit no-ops, the instrumentation code may still increase code size even when it does not increase cycle count.

Our approach goes beyond static scheduling of instrumented code. It attempts at minimal instrumentation in a user program and makes the hardware generate profile update operations and then execute them asynchronously with the user program. Its advantages include that the code size normally will not increase, and the update operations normally will not impact user code. Since the simple profile update operations are collected into a shared buffer, they can be easily pipelined to achieve the maximal execution efficiency.

8 Conclusion and Future Work

We have presented a technique that combines strengths from both the software and the hardware to efficiently collect accurate profiles. The compiler uses its powerful analysis capability to determine the profiling locations and minimize the profiling operations. The hardware uses runtime knowledge to discover free execution slots and performs profile update operations, with little impact on the user program performance. The program execution with profiling can run almost as fast as without profiling. We believe this is the first approach that attains accurate profile information with both hardware and software support. This will enable profile-driven dynamic optimizations on EPIC processors.

In the future, we would also like to simulate the profiling hardware to measure the buffer size and the memory traffic due to profile update operations. We have implemented the needed compiler support for cycle-accurate performance simulation, in which the new profiling instructions are emitted as special nops.

References

1. Anderson, J., L.M. Berc, J. Dean, S. Ghemawat, M.R. Henzinger, S.T. Leung, R.L. Sites, M.T. Vandevoorde, C.A. Waldspurger, and W.E. Weihl, “Continuous profiling: where have all the cycles gone?” In Proc. 16th Symposium on Operating System Principles, Oct. 1997.
2. Arnold, Matthew, Barbara G. Ryder, “A framework for reducing the cost of instrumented code”, Proceedings of the ACM SIGPLAN'01 conference on Programming language design and implementation, p.168-179, June 2001, Snowbird, Utah, United States.
3. August, D.I.; Connors, D.A.; Mahlke, S.A.; Sias, J.W.; Crozier, K.M.; Ben-Chung Cheng; Eaton, P.R.; Olaniran, Q.B.; Hwu, W.-M. W. “Integrated predicated and speculative execution in the IMPACT EPIC architecture”, , 1998. Proceedings of 25th Annual International Symposium on Computer Architecture, 1998, Page(s): 227 -237
4. Ball, Thomas and James Larus, “Optimally profiling and tracing programs,” ACM Transactions on Programming Languages and Systems, 16(3): 1319-1360, July 1994.
5. Ball, Thomas and James Larus, “Efficient Path Profiling,” MICRO-29, December 1996.
6. Conte, T.M., B.A. Petal, and J.S. Cox, “Using branch handling hardware to support profile-driven optimization,” In Proc. 27th Annual Intl. Symposium on Microarchitecture, Dec. 1996, pp 36-45.
7. Conte, T.M., K.N.Menezes, and M.A. Hirsh, “Accurate and practical profile-driven compilation using the profile buffer,” In Proc. 29th Annual Intl. Symposium on Microarchitecture, Nov. 1994, pp 12-21.
8. Dean, J., J.E. Hicks, C.A. Waldspurger, W.E. Weihl, and G. Chrysos, “ProfileMe: Hardware Support for Instruction-level Profiling on Out-of-Order Processors,” Micro-30, Dec. 1997.
9. Diep, Trung A., Christopher Neslson, and John P. Shen, “Performance Evaluation of the PowerPC 620 Microarchitecture. In Proceeding of the 22nd Annual International Symposium on Computer Architecture, pp 163-174, June 1995.
10. Ebcioglu, K.; Altman, E.; Gschwind, M.; Sathaye, S. “Dynamic binary translation and optimization,” IEEE Transactions on Computers, Volume: 50 Issue: 6, June 2001, Page(s): 529 -548
11. Eichenberger, A. and Sheldon M. Lobo, “Efficient Edge Profiling for ILP-Processor,” PACT 98.
12. Knuth, D. E. and F. R. Stevenson, “Optimal measurement of points for program frequency counts,” BIT 13 pp. 313-322 (1973).
13. Intel Corp, “Itanium Application Developers Architecture Guide,” May 1999.
14. Lee, Yong-fong and Barbara G. Ryder, "A Comprehensive Approach to Parallel Data Flow Analysis", Proceedings of the ACM International Conference on Supercomputing, Pages 236-247, July 1992.

15. Merten, Matthew C., Andrew R. Trick, Christopher N. George, John C. Gyllenhaal, and Wen-mei W. Hwu, “A Hardware-Driven Profiling Scheme for Identifying Program Hot Spots to Support Runtime Optimization,” Proceedings of the 26th International Symposium on Computer Architecture, May 1999
16. Merten, M.C.; Trick, A.R.; Nystrom, E.M.; Barnes, R.D.; Hwu, W.-M.W. “A hardware mechanism for dynamic extraction and relayout of program hot spots,” 2000. Proceedings of the 27th International Symposium on Computer Architecture, 2000, Page(s): 59 -70
17. Schnarr, Eric and James Larus, “Instruction Scheduling and Executable Editing,” Micro 29, Dec. 1996.
18. Schlansker, M.S., Rau, B.R. “EPIC: Explicitly Parallel Instruction Computing,” Computer, Volume: 33 Issue: 2, Feb. 2000, pp 37-45
19. Zhang, Xiaolan, Zheng Wang, Nicholas Gloy, J. Bradley Chen, and Michael D. Smith. “System Support for Automated Profiling and Optimization,” 16th ACM Symposium on Operating System Principles, Oct. 5-8, 1997.

Continuous Adaptive Object-Code Re-optimization Framework

Howard Chen, Jiwei Lu, Wei-Chung Hsu, and Pen-Chung Yew

University of Minnesota, Department of Computer Science
Minneapolis, MN 55414, USA
{chenh, jiwei, hsu, yew}@cs.umn.edu
http://www.cs.umn.edu/~hsu/dynopt

Abstract. Dynamic optimization presents opportunities for finding run-time bottlenecks and deploying optimizations in statically compiled programs. In this paper, we discuss our current implementation of our hardware sampling based dynamic optimization framework and applying our dynamic optimization system to various SPEC2000 benchmarks compiled with the ORC compiler at optimization level O2 and executed on an Itanium-2 machine. We use our optimization system to apply memory prefetching optimizations, improving the performance of multiple benchmark programs.

1 Introduction

Dynamic optimization presents an opportunity to perform many optimizations that are difficult to apply at compile time due to information that is unavailable during static compilation. For instance, dynamic link libraries limit the scope of procedure inlining and inter-procedural optimizations, two optimizations that are known to be very effective [2]. In addition, a typical shared library calling sequence includes expensive indirect loads and indirect branches. Such instruction sequences are good targets for dynamic optimizations once the shared libraries are loaded.

Dynamic optimization also provides an opportunity to perform micro-architectural optimizations. Recompiling a program to a new micro-architecture has been shown to greatly improve performance [9]. Dynamic optimization provides a way to re-optimize a program to new micro-architectures without requiring recompilation of the original source code.

Finally, dynamic optimization can specialize a program to a specific input set or user which has been applied successfully by Profile Based Optimization (PBO) [2],[5] in the past, but are difficult to apply for due to concerns over excessive compile time, instrumentation-based profiling overhead, complex build processes, and inadequate training input data set [14]. Dynamic optimization can be used to deploy more aggressive optimizations, such as predication [13], speculation [10], and even register allocation and instruction scheduling according to current program behavior and with less risk of degrading performance.

P.-C. Yew and J. Xue (Eds.): ACSAC 2004, LNCS 3189, pp. 241–255, 2004.

In short, dynamic object code re-optimization allows code to be generated specifically for a specific execution environment. It adapts optimizations to the actual execution profiles, micro-architectural behavior, and exploits the opportunity to optimize across shared libraries. However, a dynamic optimization system must detect and apply optimizations efficiently to be profitable. If the overhead of the system is greater than the time saved by optimizations, the runtime of the optimized program will increase.

We present our design and implementation of our adaptive object-code re-optimization framework to detect and deploy dynamic optimizations with a minimal amount of overhead on modern hardware. Our prototype system detects time-consuming execution paths and performance bottlenecks in several unmodified SPEC2000 benchmarks by *continuously* sampling Itanium performance monitoring registers throughout the program's execution [6]. We use the collected information to create executable traces at runtime, and deploy these optimizations by modifying branch instructions in existing code to execute our instructions in place of hot paths in the original code. We examine the overhead of our detection system, and show that this technique can be applied with less than 2% of overhead while speeding up various SPEC2000 benchmarks.

2 Background

Dynamic optimization has been presented in the past in frameworks such as Dynamo [3] and Continuous Profiling and Optimization (CPO)[12]. Dynamo uses a method similar to dynamic compilation used in virtual machines. Native binaries are interpreted to collect an execution profile and fragments of frequently executed code are emitted and executed in place of interpretation. Dynamo requires no additional information beyond the executable binary to operate, and this allows it to be applied on arbitrary binaries without needing access to the original code or IR. CPO presents a model closer to traditional PBO where the original code is instrumented, and the profile information is used to compile optimized versions of code. In CPO, profiled information is used to drive PBO while the program is running and the compiled result is hot-swapped into the program. The advantage of this scheme is that the IR information makes application of many optimizations easier.

However, since the applied optimizations compete with the dynamic optimization system's overhead, interpretation and instrumentation-based profiling dynamic optimizers often try to limit the time spent collecting profiled information, sometimes at the expense of optimizations. For example, Dynamo only interprets unique execution paths a small number of times before selecting traces to optimize to avoid interpretation overhead. Instrumentation incurs less overhead than interpretation, but even efficient implementations of instrumentation [4] generate measurable overheads. This may lead to profiling of initialization behaviors that do not represent dominant execution behavior. Even after attempting to reduce optimization system costs, the dynamic optimization systems still produce a relatively high amount of overhead, which works against the profitability of optimizations. In our system, we seek to limit the overhead of the techniques used to profile and deploy dynamic optimizations.

Existing dynamic optimization systems can generally be broken down into three stages: profiling/detection, optimization, and deployment. The detection stage deals with the collection of information necessary to select and guide optimizations. The optimization stage uses the collected information to select a set of optimizations to deploy in a target program. The deployment stage handles the application of the selected optimizations to a running program. Each of these stages requires runtime processing which leads to a slowdown of the original program. We try to reduce target profiling and deployment costs to improve the performance of the entire dynamic optimization system.

Our overall goal is to move towards a dynamic optimization framework that incurs minimal overhead while providing good potential for optimization speedups. Other techniques seek to perform similar goals [15], [16] using hardware. Like these schemes, our work uses specialized hardware to collect information useful for our planned optimizations. However, these schemes propose the implementation of new hardware to process data. In contrast, our work gathers data from existing performance monitoring hardware and analyzes it using a user program. Previous work in periodic sampling of hardware structures is presented in [1], [7], [8]. These schemes concentrate on applying this information to guide static PBO rather than dynamic optimizations.

3 Architecture

3.1 Overview

Our architecture performs three main tasks: detection, optimization, and deployment. Detection deals with the collection of raw performance event information that is useful for identifying and applying optimizations like D-cache misses, IPC, and branch paths commonly leading up to performance events. Optimization deals with generating optimized code to replace existing executable code. Deployment deals with the issues presented by redirecting execution from the original program to optimized code.

The code for our dynamic optimizer is first executed when a program executes the C run-time startup routines. We compile our own custom version of C run-time library to start a thread dedicated to dynamic optimization and initialize a shared memory area to place optimized code. The dynamic optimization thread begins monitoring the behavior of the original primary thread, and generates and deploys optimized code later in execution. After initialization of the dynamic optimization thread is complete, the C run-time library startup routines continue and begin executing the original program while the optimization thread begins detecting optimization opportunities.

3.2 Performance Event Detection

3.2.1 Performance Monitoring Hardware

We use the Performance Monitoring Unit (PMU) on Itanium processors to collect information and signal the operating system to collect and store profiled information. The primary PMU features we use for the detection work are the performance-event counters and the Branch Trace Buffer (BTB)[11]. The performance-event counters are a set of registers that track the number of times performance events like cache misses and branch mispredictions occur. These counters are used to throw interrupts periodically after a number of events occur. For instance, we can choose to throw a system interrupt once every million clock cycles, once every ten thousand D-cache misses, or once every one-hundred branch mispredictions. During each interrupt, we can save information about the type of interrupt and BTB information in memory for later processing.

The BTB, not to be confused with a branch target buffer used in branch prediction, is a set of eight registers that store the last four branches and branch targets. When the performance monitor throws an interrupt, we use the BTB to find the last four taken branch instructions and branch targets that lead up to the performance event interrupt. By monitoring only taken branches, we can form longer traces than if we had monitored all branches since not-taken branch information can easily be reconstructed by scanning the instructions between the last branch target and the next taken branch instruction. The BTB also allows us to generate an edge profile without scanning and decoding the original source code.

A detailed discussion of PMU hardware on Itanium processors can be found in [11].

3.2.2 Perfmon

Our hardware information is collected using perflib, a library from the Perfmon toolset. The Perfmon toolset configures and collects raw performance information from Itanium programs running on 64-bit Linux. Perfmon sends system calls to a Linux kernel driver to configure the PMU and automatically collects samples of the PMU registers for later processing.

Once the PMU stores raw register information to memory, it can be consumed independently from the monitored program. All information collected by Perfmon is done without modifying the original program binary. Perfmon is described in greater detail in [20].

3.2.3 Hot Trace Selection

The goal of our hot trace detector is to find a small number of traces that lead up to performance critical blocks. To collect these hot traces, we sample sets of four taken branches and branch targets from the BTB at regular intervals and sort the results into a hash table, while keeping track of the frequency of the different sample paths. The most frequently sampled paths are marked for optimization in the table under the assumption that they dominate execution time, and traces are selected to include these hot spots.

Since the optimization and deployment of traces requires processing time, we limit the traces we select to those we believe we can optimize profitably. The best traces to optimize are the traces that contribute the most execution time for the remainder of the program, and contain a performance event that we can optimize. The performance monitoring hardware provides the information we need to see if a performance event commonly occurs on any path we select. However, to predict which traces will continue to dominate execution time in the future and prevent optimizing traces with performance events that only occur for very short periods of time, we perform additional work to estimate when program behavior is most stable.

3.2.4 Phase Change Detection

We assume that programs generally enter different "phases" of execution, periods of time when characteristics like IPC, D-cache hit rate, and the current working set of hot code follow similar patterns throughout a phase [17],[18]. Since our current optimizations focus on optimizing D-cache misses and improving IPC, we use the number of D-cache misses per cycle and IPC over time as metrics to guess when the behaviors we wish to optimize are most stable.

Our goal is to select hot traces to optimize the first time we encounter a new phase and to keep it in our working set for the remainder of program execution. As long as the execution of the program remains in a stable phase, no further changes are made to the optimized traces. Changes in execution phases will be detected, and such changes will trigger further optimizations.

Every second of execution time, we measure the number of D-cache misses and IPC over the past second and compare it to previously measured values. If the D-cache and IPC values are stable for several seconds (deviating less than a set percentage), we assume that the D-cache and general program behavior is stable enough to select a set of hot traces that may execute for some time into the future, and select a set of traces from the samples in the current stable phase.

Conversely, when D-cache and IPC values fluctuate, it indicates that program behavior has changed and that new hot traces may need to be selected or existing selected traces may no longer be hot. When D-cache miss rate and IPC values deviate from the previous few seconds, we recheck our collected sampled data to look for new hot traces to add to our working set.

The weakness of this metric is that D-cache miss and IPC values are composite values for all the code that is executing in a program over an interval. This can lead to a false measurement of a stable phase since it is possible that program behavior has completely changed but averages out to similar values. In practice, we found that a stable D-cache miss rate and IPC values indicated a stable phase. We did not observe this behavior in any of our measurements of SPEC benchmarks, but it remains a possibility for other programs.

A more likely problem with this metric is that program behavior patterns commonly have sub-patterns. For instance, an outer loop may contain two inner loops: one with stable exploitable behavior, and one with unstable behavior. The stable subphase, or repeated stable behavior contained within the larger phase, can be optimized despite instability in the D-cache and IPC metrics. However, this behavior was also uncommon in the SPEC benchmarks we measured.

However, these problems indicate the potential need for deeper phase change detection to fully exploit all the stable behavior in a program that may be further supported in future work studying phase detection of programs outside of the SPEC benchmark suite.

3.3 Optimization

3.3.1 Trace Generation

Every time a set of traces is selected, executable code corresponding to each trace is assigned a type name according to its behavior. For instance, a loop is a type assigned to any trace that is completely enclosed in a trace. A subroutine is a type assigned to traces that are targeted by “br.call” instructions and end with a “br.ret” instruction, both indicators of a subroutine. Traces of different types are optimized in different ways. Data prefetching loop optimizations are particularly applied only to loop-typed traces, while inter-trace optimizations are more likely to be applied in subroutines. Code is then generated in our trace buffer for our selected traces, a shared memory area that contains our optimized executable code. The selected traces are then “cross-patched” or set to return to original code if execution leaves the path of the trace.

3.3.2 Trace Cross-Patching

Cross-patching refers to patching optimized traces to branch to other optimized traces. Ideally, once a good set of optimized traces is selected, control should rarely return to the original program. To perform cross-patching, a graph is generated with one node for each selected trace and edges to connected traces. When the code for the trace is generated, the branch instructions in optimized traces are then modified to branch to other optimized traces instead of returning to the original code.

3.3.3 Architecture Safe Optimization

Code scheduling/motion and other aggressive transformations may cause problems for preserving the original order of architecture state changes. This may create problems if a user trap handler expects to have the precise architecture state at the exception. To avoid these problems, we first attempt optimizations that do not change architecture states such as trace layout and data cache prefetching. Although data cache prefetch transformations require some temporary registers to hold prefetch addresses, we have the compiler (ORC compiler) reserve four general purpose and two predicate registers for such a purpose. Due to the large register file in the Itanium architecture, reserving a small number of registers has essentially no performance impact on the compiled code. We have verified this by comparing the performance of compiled code at various optimization levels with/without the reserved registers.

3.3.4 Data Prefetching

In our dynamic optimization system, we target D-cache misses for optimization because they are well known to be a common dominant performance problem in programs, but are difficult to detect statically [19]. Using our ability to detect instructions that cause D-cache misses, our dynamic optimization system can take advantage of information that is only available at run-time.

To detect which memory operation generates data cache misses during runtime, D-cache miss events are sampled by the performance monitoring unit and associated with the corresponding load/store instructions in the selected traces. Once a memory operation is found with a miss latency that contributes greater than 5% of execution time based on performance monitoring counters, the trace where this instruction resides will be scheduled for optimization.

The optimizer then determines if any of the implemented prefetch optimizations are appropriate. We implement three types of data prefetching for loops: array reference, indirect array reference, and pointer chasing.

Here is an example of the code generated in an indirect array reference before prefetching:

```
loop:
ld4 r17=[r43] // cache miss from loading the address stored in the array
sxt4 r17=r17
add r43=4,r43   // the array pointer is incremented
add r17=r39,r17
add r17=-1,r17
ld1 [r17]       // cache miss from loading the data value from the address
...
br loop
```

A frequent D-cache missed indirect array reference will trigger two-level data cache prefetching. The first level is a direct array reference prefetch to prefetch for the array containing the addresses of the data. The second level prefetches the value at the address from the array. Note that the first level runs a few iterations ahead of the second level of prefetching. Here is an example of the prefetch code generated to optimize the previous indirect array-reference code:

```
add  r30=128,r43  // this initializes the direct reference prefetch
add  r28=64,r43 // this initializes the indirect prefetch
loop:
lfetch [r30],4    // this prefetches the address from the array
...
```

```
ld4.sa r29=[r28],4 // this loads the prefetched address
          // (prefetched in a previous iteration)
sxt4  r29=r29
add  r29=r39,r29
add  r29=-1,r29
lfetch [r29]   // this prefetches the indirect value from the array address
...
br loop
```

This is an example of pointer chasing code before optimization:

```
loop:
add r22=24,r11 // calculate offset of pointer to next list element
ld8 r11=[r22] // cache miss from loading address of next list element
br loop
```

For pointer-chasing prefetching, the key memory address which controls the memory references (i.e.the possible "->next" pointer for linked lists) is found and prefetched by assuming a constant stride:

```
loop: (assume r28 is reserved unused by static compiler)
add  r28=0,r11    // remember the old value of r11
add  r22=24,r11
ld8  r11=[r22]
sub  r28=r11,r28   // calculate the difference of old and new value
shladd r28=r28,2,r11  // use the difference as stride to prefetch
lfetch [r28]     // prefetch ahead in the linked lists.
```

3.4 Deploying Optimizations

3.4.1 Patching Branch Targets

We redirect execution from the original code to traces in our trace buffer by modifying frequently executed branch instructions to branch to the corresponding optimized code in the trace buffer. However, modifying executable instructions in a running program creates a number of issues ranging from memory protection to updating the I-cache line with the modified instruction.

3.4.2 Memory Protection

The memory protection on the original code pages is read-only by default. When we wish to modify branch instructions in the address space of existing code, we make a system call to allow writing to memory pages of the original code. We then replace all branches at once, then restore write-protection to the original code to protect the original code from accidental changes.

3.4.3 Branch Patching Distance

Most programs compiled for the Itanium architecture use a short branch, which allows a relative branch distance of 20-bits or about a million bundles (i.e. 16 megabytes). There are cases when the original code size is larger than 16 megabytes and a branch cannot be easily patched by simply changing the address field of a branch instruction. In some cases, the entire bundle containing the branch instruction must be replaced with a long branch instruction to reach the memory space of the trace buffer. The long branch instruction in the Itanium architecture allows the branch to reach anywhere in the 64-bit virtual address space. One alternative to using a long branch is to use an indirect branch instruction sequence, but this is more expensive, more likely to be mispredicted, and is more difficult to patch atomically.

Since the Itanium uses an explicitly parallel instruction computing (EPIC) architecture, instructions are combined into 128-bit "bundles" which usually contain three instructions. Bundles with long-branch instructions can only contain two instructions. If the second instruction in a bundle that was replaced contains a "nop" in the middle slot, a common case, the entire bundle can be replaced at once. However, if the bundle you wish to replace with a long branch uses all three slots, the trace patcher cannot replace the instruction with a single instruction and we patch the target of the short branch instruction with an unconditional long branch to our optimized traces.

3.4.4 Atomic Write Issues

Another issue with replacing instruction bundles is that bundles are 128 bits long while the processor only supports 64-bit atomic write instructions. That means that we need to take steps to prevent partially modified bundles from being executed. To deal with this case, we first patch the first half of the bundle with an illegal bundle type and handle the exception if the bundle is executed before we finish patching it. We then modify the second half of the bundle with the replacement bundle, and complete the process by modifying the first half of the long-branch instruction bundle.

It is also possible that a context switch occurs while a bundle is only partially executed. This can happen if a cache miss occurs in a bundle. As long as the memory operation occurs in the first slot, this bundle can still be replaced with a long branch instruction. If the partially executed bundle is replaced with a bundle with a long branch, the bundle resumes execution at the second slot in the bundle, the long branch instruction.

3.4.5 Repatching Issues

When a phase changes, some previously optimized traces may not be "hot". In general we do not undo or remove existing patched traces for two reasons. First, when we optimize a trace, it often requires less execution time and therefore appears less “hot”. However, removing the optimization would lead to the code taking more execution time, so we are better off keeping the trace patched into the program. There are cases when we attempt to optimize a trace, and it requires more execution time than before due to additional cache misses. In the future, we plan to further explore the benefits of tracking the performance of our optimization at run-time and removing optimizations that appear to degrade performance.

Second, we found that phase behavior tends to repeat over time and previously generated traces are often used again in the future. If existing traces are no longer hot, the patched traces generally have very little performance impact. This can save some processing work to regenerate the trace if the behavior becomes hot again. For long running programs that exercise many different execution paths, this may lead to fragmentation of generated traces that may affect I-cache performance. We plan to explore the benefits of optimized trace layout management in memory in the future.

4 Experiments and Discussion

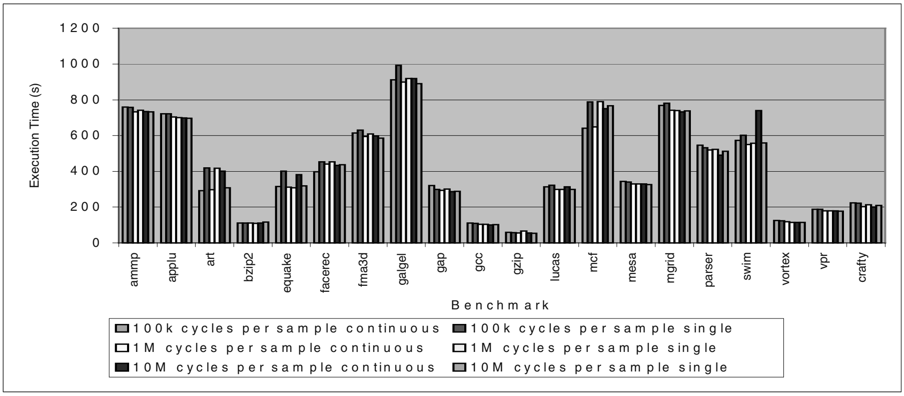
Our results are collected using ORC version 2.0 compiled SPEC2000 benchmarks with -O2 and software pipelined loops disabled on a dual processor Itanium-2 machine. We compile using O2 because that is the typical optimization level used by software vendors. Our current optimizer does not support register renaming in software-pipelined loops, so we disable software pipelining at compile time. For SPECint benchmarks, we found that disabling pipelining results in a slightly higher runtime performance when measuring results using ORC.

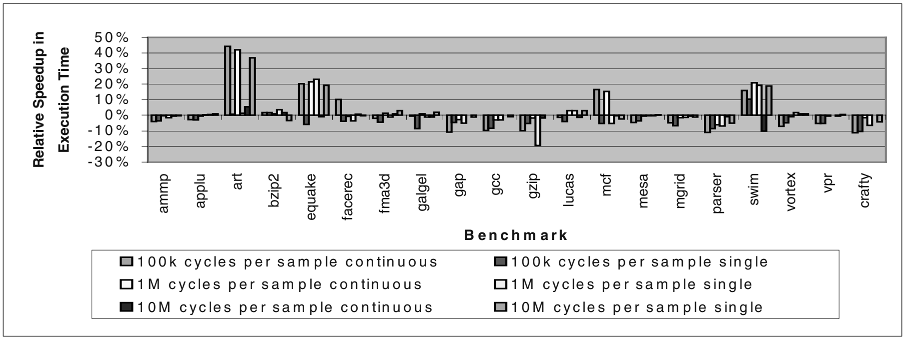
4.1 Speedup and Coverage

Execution time is collected using the unix “time” command and averaged over several runs. The reported execution time includes all detection, profiling, optimization and deployment overhead. Relative speedup is calculated as ((baseline time)/(optimized time) - 1) * 100%.

Figures 1 and 2 show the speedup of applying our system to various spec programs. We apply different sampling rates to collect data and select new traces every second of execution time. Data that is “continuous” selects new phases upon a suspected phase change while “single” data selects the first stable phase identified in a program.

As Figure 3 shows, art, bzip, equake, fma, galgel, lucas, mcf, and swim benefit from regular and indirect array reference prefetching. Mcf benefits primarily from pointer reference prefetching. In Figure 2, at one hundred thousand cycles per sample facerec speeds up by 10%. Although the D-cache miss rate of facerec appears to increase in Figure 5, the actual execution time of the program decreases. The D-cache miss rate increases but these misses overlap with each other more effectively than in the original program leading to improved program performance. In contrast, equake’s D-cache performance is noticeably improved in Figure 4. Lower sampling rates for





Figs. 1 (top) and 2 (bottom). Figure 1 shows the execution time of dynamically optimized programs with data collected at different sampling rates. Figure 2 shows the relative speedup at different sampling rates

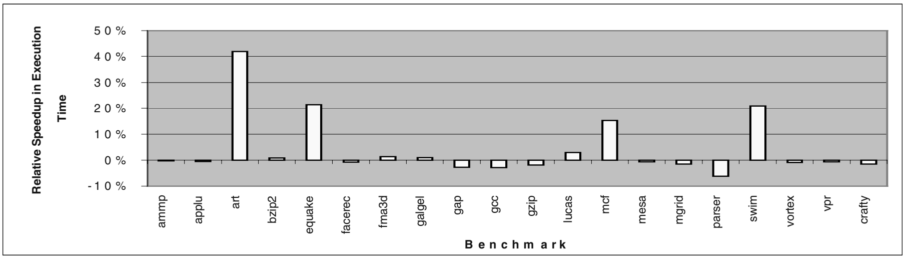
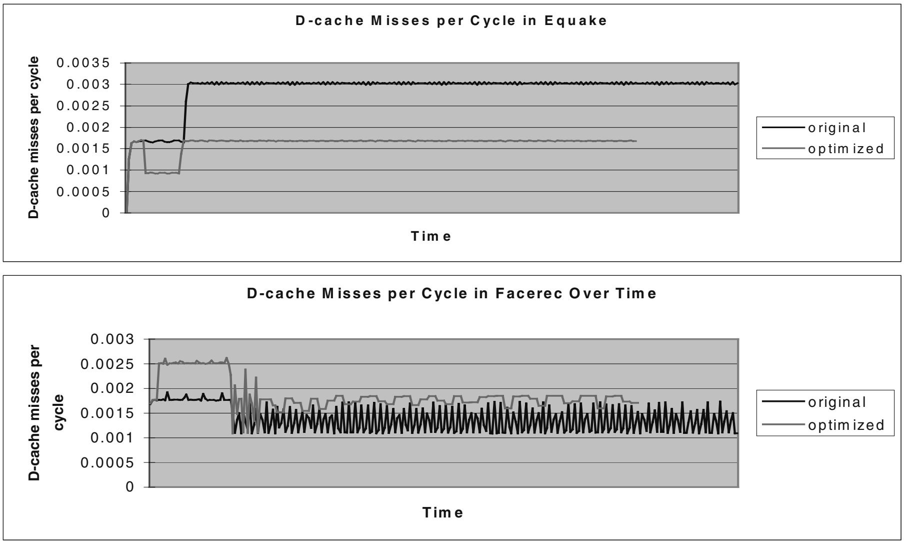


Fig. 3. Relative speedup at one million cycles per sample

facerec do not improve performance because at slower sampling rates the primary execution path is not optimized properly on the first pass and since we do not currently monitor the performance of generated traces, the problem is not corrected in future intervals. Although this problem is most dramatic in facerec, this trend can be seen in all the programs at a slower sampling rate. It may be valuable to study tracking and removal of sub-optimal traces in the future to deal with this problem.



Figs. 4 and 5. Figure 4 shows the D-cache misses per cycle in Equake before and after optimization at one sample every 100k cycles. Figure 5 shows the D-cache misses per cycle in Facerec before and after optimization at one sample every 100k cycles.

Other benchmarks do not benefit from the implemented prefetching optimizations and are primarily slowed down by the overhead of sampling, additional I-cache misses from branching to our generated traces, and increased D-cache misses due to ineffectual prefetches. The overhead of our optimization and patching is generally very small, less than 1% of the original program's execution time, so the majority of the slowdown in programs can be attributed to these factors. The largest reported slowdown is from gzip due to failed prefetching increasing D-cache misses. This demonstrates the need for additional work in tracking the effectiveness of optimizations and removing optimizations that fail to improve performance.

In general, increasing the sampling rate results in higher overhead due to the time required to store sampled PMU information. However, it also detects a set of hotspots faster than slower sampling rates, which means hot traces may be applied earlier to a program than at slower sampling rates. Mcf and art both have small loops that noticeably speed up after being optimized and therefore benefit from higher sampling rates. In contrast, equake and swim perform better at one sample taken every one million cycles. At one sample every one million and hundred thousand cycles, these two programs generally found similar hotspots, but at one-hundred thousand cycles per sample, the overhead of sampling is about 5% higher than at one sample every 1 million cycles.

Using a rate of ten million cycles per sample, continuously selecting traces yields worse performance than selecting a single set of traces once. At a rate of ten million cycles per sample, the system starts with correctly selected paths and later selects suboptimal paths that degrade performance. Because traces occur after a fixed interval, one second, the sampling error from the small pool of samples falsely detects traces as hot. Since the sampling error of a program is related to the size of the footprint we need to detect, this indicates that it might be worthwhile to estimate the size of the working set and adjust the number of samples required to select hot-spots accordingly. However, performance is improved for continuously selected traces due to the ability to select new hot traces that do not occur earlier in execution.

The largest slowdown in figure 3 is parser at 5%. This is mainly due to generated trace overhead from the selection of a large number of traces, and the lack of any effective D-cache prefetching. However, other programs like gap, gcc, and gzip generate less than 2% overhead.

Finally, in some cases selecting a set of traces after the first detected hot phase performed better than continuous selection. Continuous selection is sensitive to short term behavior changes leading the optimizer to generate more traces and more overhead than making a single selection.

4.2 Run-Time Overhead

The overhead for our run-time system is fairly stable, with the profiling thread generating a consistent 0.3%-0.6% overhead over no profiling. Optimization overhead is proportional to the number of traces selected to optimize and consistently less than 1%. Turning off patching has a negligible effect on overhead indicating that the cost of patching traces is much smaller than the cost of profiling and optimization.

Sampling overhead averaged approximately 4% at one sample every hundred thousand cycles, about 1% at one sample every million cycles, and much less than 1% at lower sampling rates. The overhead is not directly proportional to the sampling rate because this includes the overhead of inserting branch information into the trace selection table. Slowdowns of greater than 1% are primarily due to optimizations resulting in larger loops.

5 Summary and Future Work

Dynamic optimization promises to provide a useful mechanism for deploying aggressive optimizations targeting run-time behavior. We present our system as a prototype for finding and deploying optimizations, and support this claim by using our prototype to speedup various SPEC2000 benchmarks compiled by the ORC 2.0 compiler at O2. We are able to speed up several benchmarks dominated by D-cache misses, while maintaining a maximum slowdown of 5% in parser and crafty.

Future directions for this work include enhancements to our current system, monitoring current optimizations and tracking the performance of generated traces, improving phase detection, evaluating other optimization techniques, and exploring dynamic optimization opportunities in different environments.

References

1. J.M. Anderson, L.M. Berc, J. Dean, S. Ghemawat, M.R. Henzinger, S.A. Leung, R.L. Sites, M.T. Vandevoorde, C.A. Waldspurger and W.E. Weihl. “Continuous profiling: where have all the cycles gone?” *ACM Transaction on Computer Systems*, vol. 15, no. 4, Nov. 1997
2. A. Andrew, S. De Jong, J. Peyton, and R. Schooler "Scalable Cross-Module Optimization", In *Proceedings of the ACM SIGPLAN '98 conference on Programming language design and implementation*, PLDI'98, June 1998.
3. V. Bala, E. Duesterwald, S. Banerjia. “Dynamo: A Transparent Dynamic Optimization System”, In *Proceedings of the ACM SIGPLAN '2000 conference on Programming language design and implementation*, PLDI'2000, June 2000.
4. Ball, T., and Larus, J. R. “Efficient Path Profiling,” In *Proceedings of the 29th Annual International Symposium on Microarchitecture* (Micro-29), Paris, 1996.
5. P. Chang, S. Mahlke and W. Hwu, "Using Profile Information to Assist Classic Compiler Code Optimizations," *Software Practice and Experience*, Dec. 1991.
6. H. Chen, W. Hsu, J. Lu, P. -C. Yew and D. -Y. Chen, “Dynamic Trace Selection Using Performance Monitoring Hardware Sampling”, International Symposium on Code Generation and Optimization, CGO 2003, March, 2003.
7. R.S. Cohn, D.W. Goodwin, P.G. Lowney, “Optimizing Alpha Executables on Windows NT with Spike”, *Digital Technical Journal*, Vol 9 No 4, June 1998.
8. T. Conte, B. Patel, J Cox. “Using Branch Handling Hardware to Support Profile-Driven Optimization”, *In Proceedings of the 27th Annual International Symposium on Microarchitecture* (Micro-27), 1994
9. A. M. Holler, “Optimization for a Superscalar Out-of-Order Machine,” *In Proceedings of the 29th Annual International Symposium on Microarchitecture* (Micro-29), December 1996.
10. Intel, *Intel IA-64 Architecture Software Developer’s Manual*, Vol. 1: IA-64 Application Architecture.
11. Intel, *Intel IA-64 Architecture Software Developer’s Manual*, Vol. 2: IA-64 System Architecture.
12. T. Kistler, M. Franz. “Continuous Program Optimization: Design and Evaluation”, *IEEE Transaction on Computers*, vol. 50, no. 6, June 2001.
13. S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann, “Effective Compiler Support for Predicated Execution Using the Hyperblock”, *In Proceedings of the 25th Annual International Symposium on Microarchitectures*. (Micro-25), 1992 .
14. S. McFarling, “Reality-Based Optimizations”, International Symposium on Code Generation and Optimization, CGO 2003, March, 2003.
15. M. Merten, A. Trick, E. M. Nystrom, R. D. Barnes, W. Hwu, “A Hardware Mechanism for Dynamic Extraction and Relayout of Program Hot Spots”, *In Proceedings, International Symposium on Computer Architecture*, ISCA-27, 2000
16. S. Patel, S. S. Lumetta, “Replay: A Hardware Framework for Dynamic Optimization”, *IEEE Transaction on Computers*, vol. 50, no. 6, June 2001.
17. T. Sherwood, E. Perelman, G. Hamerly, B. Calder. “Automatically Characterizing Large Scale Program Behavior. *10th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 2002.
18. T. Sherwood, E. Perelman, and B. Calder. Basic block distribution analysis to find periodic behavior and simulation points in applications. *In International Conference on Parallel Architectures and Compilation Techniques*, September 2001.

19. Y. Wu. “Efficient Discovery of Regular Stride Patterns in Irregular Programs”, PLDI 2002:210-221.
20. Hewlett Packard, “Perfmon Project Website”, webpage, http://www.hpl.hp.com/research/linux/perfmon/index.php4.

Initial Evaluation of a User-Level Device Driver Framework

Kevin Elphinstone[1] and Stefan Götz[2]

[1] National ICT Australia[***]
and
School of Computer Science and Engineering
University of NSW, Sydney 2052, Australia
kevine@cse.unsw.edu.au

[2] System Architecture Group, Universität Karlsruhe, 76128 Karlsruhe, Germany
sgoetz@ira.uka.de

Abstract. Device drivers are a significant source of system instability. In this paper, we make the case for running device drivers at user-level to improve robustness and resource management. We present a framework for running drivers at user-level whose goal is to provide similar performance when compared to in-kernel drivers. We also present initial promising performance results for the framework.

1 Introduction

Most modern operating systems feature monolithic operating system kernels. Most modern architectures are designed to efficiently support this form of construction. A kernel provides its services by combining the software that implements potentially independent services into a single large amalgamation. However, once we scale the size and complexity of a monolithic system to the levels of current systems, extensibility becomes more difficult due to legacy structure, security becomes more difficult to maintain and impossible to prove, and stability and robustness also suffer.

One promising approach to tackling the expanding complexity of modern operating systems is the microkernel approach [1]. A microkernel-based OS consists of a very small kernel at its core. The kernel only contains a minimal set of services that are efficient and flexible enough to construct services for applications as servers running on the microkernel. Only the microkernel itself runs in privileged mode. Although these servers provide operating system functionality, they are regular applications from the microkernel's point of view. Such a system enables extensibility as servers can be added or removed, it provides security as the core of the system is small enough to analyse or maybe even prove [2], and

[***] National ICT Australia is funded by the Australia Government's Department of Communications, Information and Technology and the Arts and the Australian Research Council through Backing Australia's Ability and the ICT Centre of Excellence program.

P.-C. Yew and J. Xue (Eds.): ACSAC 2004, LNCS 3189, pp. 256–269, 2004.

stability and robustness is improved as services can be isolated from each other. The modular structure that is encouraged, and even enforced by virtual memory protection boundaries, improves maintainability.

Most microkernel based systems still include device drivers in the kernel. Drivers are included for either security [3], performance reasons [4] , or because the system's focus was toward goals other than decomposition and minimisation, such as distribution [5,6,7]. It has been shown that device drivers exhibit much higher bug rates (three to seven time higher) than other kernel code [8]. Microsoft has also identified drivers as the major cause of system instability and has instigated their driver signing program to combat the problem [9]. It remains to be seen whether signing a driver as having passed a quality control scheme has an affect on driver correctness. Simply digitally signing a piece of software obviously has no effect on the software itself.

This paper tackles the problem of device driver instability by running drivers at user-level and hence subjecting them to the normal controls applied to applications. We also aim to provide a flexible driver framework for microkernel-based systems that enables trade-offs between driver performance and containment. Attempts thus far can be characterised as being too concerned with compatibility with existing driver collections [10] or having an alternative focus such as realtime systems [11]. The achieved performance has been insufficient to make the approach convincing.

Device drivers at user level could be treated almost like normal applications. Like normal applications, drivers could be isolated from unneeded resources using the processor's virtual memory hardware. Being able to apply the principle of *least privilege* would greatly minimise the potential damage a malfunctioning driver could inflict. This is very much in contrast to the current situation where drivers have access to all resources in the system. A single malfunction often results in catastrophic failure of the entire system.

Developers of user-level drivers can use facilities usually only available to normal applications. Standard debuggers can provide a much richer debugging environment than usually available to kernel-level drivers (e.g., source level debugging versus kernel dumps). Application tracing facilities can also be used to monitor driver behaviour. Application resource management, such as CPU time controls, can be used to control driver resource usage.

User-level drivers are not a completely new idea. Drivers in the past have been incorporated into applications such as networking software (e.g distributed shared memory applications) [8,12]. The inclusion in this case was to improve performance by giving the application direct access to the device, and thus avoiding kernel entry and exit. Such scenarios relied on near exclusive access to the device in order to avoid issues in multiplexing the device between competing clients. In most cases, specialised hardware was developed to provide concurrent access via specialised access channels, and to provide performance via a specialised interface that required no kernel intervention.

We propose an architecture where the system designer can choose the most appropriate configuration for drivers based on requirements of the targeted system. We envisage drivers incorporated into specialised applications where performance is paramount. However, we also envisage drivers running as individual servers to improve security and robustness, or drivers clustered into a single server to reduce resource requirements. Immature drivers could be run in isolation until mature enough to be combined with other components when required.

While we intend to take advantage of specialised hardware (such as myrinet network cards which have their own programmable processors [13]), we also do not intend to restrict ourselves to such hardware. For the results of this project to be truly useful we must be able to support commodity hardware that is not necessarily tailored to the environment we are developing. Commodity hardware may not provide all features necessary for complete security. For instance, nearly all hardware is unable to restrict what a driver can access via DMA. On such a hardware platform, a malicious driver can always corrupt a system. However, even limited success in supporting commodity hardware with little performance impact would make our results applicable to the widest variety of platforms possible. Limited success could persuade more manufacturers to include the hardware features required for complete security. Our group has begun exploring restricting DMA access using the limited hardware available in high-end servers [14]. However, we do not focus on this problem for the remainder of this paper.

Past approaches to drivers at user-level have usually taken a top-down approach. The system was designed with a specific target in mind, built, and analysed. The results have varied widely. Some projects, specifically the user-level networking with specialised hardware, have been successful [12,15]. Other projects have been less successful and have usually disappeared without a clear analysis of why success eluded them [10,16]. In this paper we identify the fundamental operations performed by device drivers, their relevance to performance, and present how they can be implemented safely and efficiently at user-level.

In the remainder of the paper, Section 2 provides the background to running user-level drivers by describing a simple model of device drivers in existing monolithic systems. We use it as a reference for the rest of the paper. Section 3 describes the experimental operating system platform upon which we developed our driver framework. Section 4 describes the framework itself. The experimental evaluation and results follow in Section 5, with conclusions afterwards in Section 6.

2 Simple Driver Model

To define common terminology, help convey the issues we have identified, and introduce our framework itself, we present a simple model of a device driver and highlight the issues within that model. This model initially assumes a traditional monolithic kernel whose kernel address space is shared between all process contexts.

A driver broadly consists of two active software components, the *Interrupt Service Routine* (ISR) and the *Deferred Processing Component* (DPC). We ignore initialisation code and so forth. The ISR is responsible for reacting quickly and efficiently to device events. It is invoked almost directly via a hardware defined exception mechanism that interrupts the current flow of execution and enables the potential return to that flow after completion of the ISR. In general, the length of the ISR should be minimised so as to maximise the burst rate of device events that can be achieved, and to reduce ISR invocation latency of all ISRs (assuming they are mutually exclusive).

The ISR usually arranges for a DPC to continue the processing required to handle the device event. For example, a DPC might be an IP stack for a network device. A DPC could also be extra processing required to manage the device itself, or processing required to complete execution of a blocked kernel activity. Another way to view a DPC is that it is the kernel activities made runnable as a result of the execution of the ISR. It may be a new activity, or a previously suspended activity. DPCs are usually activated via some kernel synchronisation primitive which makes the activity runnable and adds it to the scheduler's run queue.

2.1 Driver Interfaces and Structure

A driver consists of an interface in order for clients (other components in the kernel) to direct the driver to perform work. For instance, sending packets on a network device. Drivers also expect an interface provided by the surrounding kernel in order to allocate memory, activate DPCs, translate virtual addresses, access the device information on the PCI bus, etc. We believe the following interfaces are important to driver performance:

Providing work to the driver. Drivers provide an interface for clients to enqueue work to be performed by the device. This involves passing the driver a work descriptor that describes the work to be performed. The descriptor may be a data structure or arguments to a function call. The work descriptor identifies the operation and any data (buffers) required to perform the work. Drivers and clients share the kernel address space which enables fast transfer (by reference) and access to descriptors and buffers.

DPCs and offloading work. Drivers also produce work for clients. A common example is a network driver receiving packets and therefore generating work for an IP stack. Like enqueueing work for the driver itself, an efficient mechanism is required for the reverse direction to enqueue work for, and activate, a DPC such as an IP stack. Work descriptors and buffers can be handled in a similar manner to enqueueing work for the driver, i.e. descriptors and buffers can be transferred and accessed directly in the kernel's address space.

Once work is enqueued for a DPC, the DPC requires activation via a synchronisation primitive. Again, the primitive can rely on the shared kernel address space to mark a DPC runnable and place it on the appropriate scheduler queue.

Buffer allocation. The buffers containing the data that is provided to the driver must be allocated prior to use and deallocated for reuse after processing. Buffers may be produced by a client and consumed by a driver (or vice versa) and are managed via a memory allocator (e.g. a slab allocator) in the shared kernel address space.

Translation. Buffers specified by user-level applications are identified using virtual addresses. DMA-capable devices require these addresses to be translated into a physical representation. This translation can be done simply and quickly by the device driver by accessing the page tables stored within the kernel address space. Additionally, some driver clients deal only with the kernel address space and can use physical addresses directly (or some fixed offset).

Pinning. DMA-capable devices access physical memory directly without any mediation via a MMU (though some architectures do possess I/O MMUs). Coordination between the page replacement policy and the device driver is required to avoid the situation where a page is swapped out and the underlying frame is recycled for another purpose while an outstanding DMA is yet to complete. Preventing pages from being swapped out is generally termed *pinning* the page in memory. This can be implemented with a bit in the frame table indicating to the page replacement algorithm that the frame is pinned.

Validation. Validation is the process of determining whether a request to the driver is permitted based on knowledge of the identity of the requester and the parameters supplied. Validation is simple when a client issues a request to a driver in a shared address-space kernel. The driver can implicitely trust the client to issue sensible requests. It only needs to check the validity of a request for robustness reasons or debugging. If needed, the client module in the kernel is usually responsible for the validity of any user-level supplied buffers or data which needs to reside in memory accessible to the user-level application. Such a validation is simple and inexpensive to perform within a shared address-space kernel — all the data required to validate an application's request is readily available.

It is clear that the model envisaged by computer architects is a fast hardware-supported mechanism to allow privileged drivers to respond to device events, and that the drivers themselves have cheap access to all the information required to perform their function via the privileged address space they share with the kernel. The high degree of integration with the privileged kernel allows drivers to maximise performance by minimising overheads needed to interact with their surrounds. This high degree of integration is also the problem: drivers detrimentally affect security, robustness, and reliability of the entire system.

3 Experimental Platform

We chose the L4 microkernel as the experimental platform for developing and evaluating our driver framework[1]. L4 is a minimal kernel running in privileged mode. It has two major abstractions: threads and address spaces. Threads are the unit of execution and are associated with an address space. A group of threads within an address space forms a task. Threads interact via a very light weight synchronous interprocess communication mechanism (IPC) [17].

L4 itself only provides primitive mechanisms to manage address spaces. Higher-level abstractions are needed to create a programming environment for application developers. The environment we use is a re-implementation of a subset of the SawMill multi-server operating system developed for L4 at IBM [10], called *Prime*. The most relevant component to this paper is the virtual memory framework [18], which we will briefly introduce here.

Dataspaces are the fundamental abstraction within the VM framework. A dataspace is a container that abstracts memory objects such as files, shared memory regions, frame buffers, etc. Any memory that is mappable or can be made mappable can be contained by a dataspace. For a thread to access the data contained in a dataspace, the dataspace is *attached* to, i.e. mapped into, the address space. Address spaces are constructed by attaching dataspaces including application code and data, heap and stack memory.

Dataspaces themselves are implemented by *Dataspace Managers*. Any task within the VM framework can be a dataspace manager by implementing the dataspace protocol. For example, a file system dataspace manager provides files as attachable dataspaces by caching disk contents within its address space, and using the underlying L4 mechanisms to map the cached content to clients who have the dataspace attached. Dataspace managers map pages of dataspaces to clients in response to the page fault handling mechanism which forwards page faults on attached dataspaces to the appropriate dataspace manager that implements the dataspace.

The dataspace and dataspace manager paradigms provide a flexible framework of object containers and object container implementors. Few restrictions are placed on participants other than implementing the defined interaction protocol correctly. However, while clients with attached dataspaces see a logical container, device drivers interacting with such a container require more information about the current dataspace state for DMA purposes. In particular, they have to know the translation between dataspace addresses and physical memory which is only known by the dataspaces' manager. In a traditional system we have the kernel implemented page tables as a central authority for translation information. With our VM framework, translation information is distributed amongst dataspace managers which creates the problem of efficient information retrieval. We describe our solution to this problem in Section 4

4 Driver Framework

As described in Section 2, the high degree of hardware and software integration in classic system architectures creates an environment for efficient driver implementation. The challenge is to keep the high level of integration when transforming drivers into user-level applications while enforcing protection boundaries between them and the surrounding system. There are obviously trade-offs to be made between the strength of the protection boundary and the cost of interacting across it. A network driver interface that copies packets across protection boundaries provides greater packet integrity and poorer performance compared to an interface that passes packets by reference. In choosing trade-offs for this paper we focused on maximising performance while still improving robustness. Drivers and their clients may corrupt the data they produce and consume, but should not be able to corrupt the operation of each other. However, our framework is not restricted to the particular trade-offs we made for this paper. A system designer can increase or decrease the degree of isolation between clients and drivers by small changes in interfaces, their implementation, or the composition of drivers and clients.

For this paper we took the following approach:

– Minimise the cost of interaction between clients and drivers by interacting via shared memory instead of direct invocation where possible. This sharing is secure in that it is done such that clients cannot interfere with the operation of drivers and vice versa. However, data buffers can be modified by clients or drivers at any point in the interaction.
– Minimise the cost of any overhead we must insert between clients and drivers (or between drivers and the kernel) to support interaction across protection boundaries.
– For any overhead that we must insert to enable interaction, we attempt to amortise the cost by combining operations or event handling where possible.

We now describe how we applied our approach to constructing a driver framework with reference to the model introduced in Section 2.

4.1 Interrupts

Direct delivery of interrupts to applications is not possible on current hardware. A mechanism is required for an ISR within a driver application to be invoked. We use the existing model developed for L4 where interrupts are represented as IPCs from virtual *interrupt threads* which uniquely identify the interrupt source. The real ISR within the kernel masks the interrupt, transforms the interrupt event into an IPC message from the interrupt thread which is delivered to the application's ISR. The blocked ISR within the application receives the message, unblocks, and performs the normal ISR functionality. Upon completion, the driver ISR sends a reply message to the interrupt thread resulting in the interrupt source being unmasked. The ISR can then block waiting for the next interrupt IPC.

While L4 IPC is very light-weight, it is not "free". We add a small amount of direct overhead to implement this clean model of interrupt delivery. Indirect overhead is incurred by context switching from an existing application to the driver application upon interrupt delivery. We expect this overhead to be low compared to the high cost of going off-chip to manage devices, and plan to reduce the overall overhead by using interrupt hold-off techniques currently applied to limit the rate at which interrupts are generated.

4.2 Session-Based Interaction

Copying data across protection boundaries is expensive. Where possible, we use shared memory to pass data by reference, or to make control and metadata information readily available to clients and drivers. Establishing shared memory is also an expensive operation both in terms of managing the hardware (manipulating page tables and TLB entries), and in terms of performing the book-keeping required in software. To amortise the cost of setting up shared memory we use a session-based model of interaction with drivers.

A *session* is the surrounding concept within which a sequence of interactions between client and driver are performed. It is expected that a session is relatively long lived compared to the duration of the individual interactions of which we expect many within a session. To enable pass-by-reference data delivery, one or more dataspaces can be associated with a session for its duration. Dataspaces can contain a shared memory region used to allocate buffers, a client's entire address space, or a small page-sized object. There are obviously trade-offs that can be made between cost of establishing a session, and the size and number of dataspaces associated with a session. To avoid potential misunderstanding, there can be many underlying sessions within our concept of a *session*. For example, an IP stack has a *session* with the network device driver through which many TCP/IP sessions can be managed.

4.3 Lock-Free Data Structures

There are obvious concurrency issues in managing data structures in shared memory. We make heavy use of lock-free techniques to manage data structures shared between drivers and their clients. We use lock-free techniques for predominately two reasons: to avoid external interaction and to avoid time-outs and recovery on locks.

Enqueueing work (packet/command descriptors and similar metadata) for a driver by explicitly invoking it requires at least two context switches per enqueued item. This would cause the high level of integration achieved in normal systems to be lost. Lock-free queues (implemented with linked lists or circular buffers) allow work to be enqueued for a driver (or a client) without requiring explicit interaction with the driver on every operation. This encourages a batching effect where several local lock-free operations follow each other, and finally the recipient driver is notified via explicit interaction (a queued-work notify event).

Lock-free techniques allow us to avoid dealing with excessive lock holding times. It is much easier to validate potentially corrupt data in a lock-free queue that is caused by a misbehaving client (we have to validate client provided data anyway), than to determine if a client is misbehaving because a lock is found held.

4.4 Translation, Validation, and Pinning

Drivers process work descriptors which can contain references to the actual buffers to be processed. Buffers are specified as ranges of addresses within dataspaces. The dataspaces are associated with the surrounding driver-client session. The dataspaces themselves are implemented by other applications (dataspace managers). This creates an interesting problem. The knowledge of a dataspace's existence, who is accessing it, and what physical frames implement it at any instant in time is known by the dataspace manager implementing the dataspace, not the client using the dataspace, and not the driver accessing the dataspace to process the requests of the client. In a traditional system, this information (page tables and frame tables) is readily available to the driver within the kernel address space. Ideally, we would again like to safely replicate the high degree of integration between driver, clients, and information required to operate.

The validation of buffers specified by the client within the above framework is simple. Given buffers are ranges of addresses within dataspaces, validation is a matter of confirming the dataspace specified is associated with the session between the driver and client.

The translation of dataspace pages to physical frames is required by drivers of DMA-capable devices. This translation is only known by a dataspace manager. Our approach thus far has been to avoid external interaction by the driver as much as possible, however translation requires this interaction in some form. To enable translation, the dataspace manager provides a shared memory region between it and the device driver: the translation cache. The translation cache is established between the manager and driver when a dataspace is added to a session between the driver and client. Multiple dataspaces from the same manager can share the same translation cache. The translation cache contains entries that translate pages within dataspaces into frames[1]. The cache is consulted directly by the driver to translate buffer addresses it has within dataspaces to physical addresses for DMA. After the translation cache is set up, the driver only needs to interact with the object implementor in the case of a cache miss. At present we use a simple on-demand cache refill policy, but we plan to explore more complex policies if later warranted.

In addition to translating a buffer address to a physical address for DMA, the driver needs a guarantee for the duration of DMA that the translation remains valid, i.e. the page (and associated translation) must remain *pinned* in memory. In this paper we have not focused on the problem of pinning in depth. We see at least two approaches to managing pinning for DMA. The first method is to use time-based pinning where entries in the translation cache have expiry times. The second method is to share state between the driver and dataspace implementor to indicate the page is in use and should not be paged out.

Time-based pinning has the difficult problem of the driver needing to estimate how long a DMA transaction might take, or even worse, how long it will take for a descriptor in a buffer ring to be processed, e.g. on a network card. However, time-based pinning has the nice property of not requiring interaction between

[1] In our virtual memory framework, dataspaces can also be composed of other dataspaces. In this case, the translation consist of a sequence of dataspace to dataspace translations, and then a final dataspace to physical frame translation. However, we ignore this scenario for the sake of clarity in the paper.

driver and object implementor. Further discussion of time-based pinning can be found our previous work [19].

State sharing to indicate to the dataspace manager that pinning is required could be achieved with a pin-bit within translation cache entries. This requires read-write shared memory between driver and dataspace implementor that was not required up until this point. It should be clear that the pin-bit has direct parallels with similar flags in a traditional frame table and thus warrants little further discussion. Note that the pin-bit would only be advisory. The memory implementor can enforce quotas on pin time or the amount of pinned memory by disabling the driver and resetting the device (if permitted) to recover pinned pages.

4.5 Notification

Unlike traditional systems where thread state and scheduler queues are readily available in shared kernel space, in a system with drivers in separate protection domains, system calls must be performed to manipulate the scheduler queues, i.e. block and activate threads. System calls are significantly more expensive than state changes and queue manipulations. An efficient activation mechanism is required for ISRs to hand-off work to DPCs, and for both clients and drivers to deliver work and potentially block as the sender and while activating the recipient.

By using queues in shared memory for message delivery, we create the environment required for user-level IPC (as opposed to IPC involving the kernel). User-level IPC has been explored by others [20,21], mostly in the context of multiprocessors where there is an opportunity to communicate without kernel interaction via shared memory between individual processors. Our motivation is two-fold. We wish to avoid kernel interaction (not activate the destination) if we know the destination is active (or will become active), and we wish to enable batching of requests between drivers and clients by delaying notifications when possible and desirable.

Our notification mechanism is layered over L4 IPC. Blocking involves waiting for a message, activating involves sending a message. To avoid notifications when unnecessary, the recipient of notifications indicates its thread state via shared memory. If marked inactive, a notification is sent; if not it is assumed that the recipient is (or will be) active and the notification is suppressed.

The delay between setting the state and blocking waiting for IPC creates a race condition if preempted between the modification and blocking waiting for IPC. There is a potential for notification messages to be missed if sent to a thread that has not yet blocked. However, if the sender does not trust the recipient, it is not safe for the sender to block on or re-send notifications without being vulnerable to denial-of-service attacks. Thus, recipients have to be able to recover from missed notifications on their own. We resolve this race by using a general mechanism called *preemption control*, which can make threads aware of preemption. In the rare case that a preemption is detected, the recipient rolls back to a safe active state from where it tries to block again.

The notification bit creates opportunities for delaying notification (to increase batching) or avoiding notification altogether. An example of avoidance is where a network driver would eventually receive a “packet sent” or “transmit queue empty” interrupt from the device. If such events are known to occur within acceptable latency bounds, notifying such a device when enqueueing an outgoing packet is unnecessary as the driver will eventually wake via the interrupt to discover the newly enqueued packets. This allows a driver client to submit requests continuously to maximise the batching effect.

5 Evaluation and Results

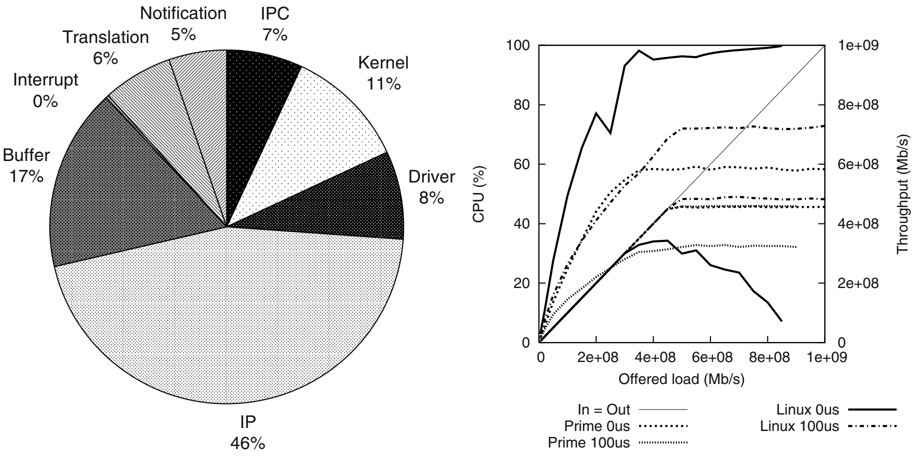
We evaluate our framework for running device drivers at user level in a network context. Handling modern high-speed networks is challenging for traditionally structured systems due to the very high packet rate and throughput they achieve.

Our test system consists of a user-level ISR that is comprised of generic low-level interrupt handling in the L4 kernel and device-specific interrupt handler for a dp83820 Gigabit ethernet card driver. The DPC is the lwIP IP stack and a UDP echo service that simply copies incoming packets once and echoes them to the sender. The driver and lwIP execute in separate processes which interact as described in Section 4. Note that the echo service is compiled into the process containing lwIP. The machine is a Pentium Xeon 2.66 GHz, with a 64-bit PCI bus.

We chose this test scenario as we believe it to be the extreme case that will expose the overheads of our framework most readily. The test does very little work other than handle interrupts, and send/receive packets across a protection boundary between the driver and lwIP, and then onto (or from) the network. In a more realistic scenario, we would expect the “real” application to dominate CPU execution compared to the drivers and IP stack. By removing the application, the driver and lwIP stack (and our overheads) will feature more prominently.

We used the ipbench network benchmarking suite[22] on four P4-class machines to generate the request UDP load that we applied to the test system. ipbench can apply specific load levels to the target machine, the packet size used was 1024 bytes. We performed two experiments, (a) that uses random-interval program counter sampling to develop an execution profile at an offered load of 450Mb/s using 100us interrupt hold-off, and (b) which ramps up the offered load gradually and records throughput and CPU utilisation at each offered load level. The load generators record echoed packets to calculate achieved throughput, CPU utilisation is measured by using the cycle counter to record time spent in a low priority background loop. Utilisation results obtained via random sampling and the cycle counter agree within one percent. The second experiment was performed for both 0μs and 100μs interrupt hold-off for Prime, and to compare, Linux with our driver and Linux’s IP stack in-kernel, and user-level echo server.

The results show that for the profile experiment 68% of time was spent in the idle thread. For the remaining 32% of samples, we divided the samples into the following categories: *IPC* kernel code associated with the microkernel IPC path; *Driver* code associated with the network driver that would be common to all drivers for this card independent of whether it runs in-kernel or at user-level; *IP* code associated with lwIP that is also independent of running at user-level or in kernel; *Kernel* code that is independent of system structuring, this code is mostly related to interrupt masking and acknowledgement; User-level *Notification* code that implements notification within our framework; User-level *Translation* code that performs translation from dataspace addresses to physical memory; User-level *Interrupt* code related to interrupt acknowledgement; User-level *Buffer* code for managing packet buffers, including (de-)allocation, within our framework.



(a) (b)

Fig. 1. (a) Execution profile. (b) CPU utilisation and throughput versus load level for Prime and Linux using 0μs and 100μs interrupt hold off.

The profile of execution within these categories is illustrated in Figure 1. The component of execution unrelated to our framework (Kernel + Driver + IP) forms 65% of non-idle execution time. Code related to our framework (Buffer + Interrupt + Translation + Notification + IPC) forms the remaining 35% of non-idle execution. Even when considering all framework related code as overhead introduced by running drivers at user-level, this is not a bad result. The test scenario we chose to analyse does so little work that we expect in a more realistic scenario our framework will consume a smaller fraction of execution time.

Considering all framework related code as overhead is not a fair comparison as two components of the framework (buffer management and translation) also have to be performed in a traditional system structure. If traditional buffer management and translation is comparable, then the overhead of running drivers at user-level (Notification + IPC + Interrupt) is only 12% of non-idle time.

Figure 1 also shows the result of the CPU utilisation and throughput experiment. The thin diagonal line represents where achieved throughput equals offered load. The lines beginning and rising above this reference represent CPU utilisation. The lines that track the reference and diverge to the right represent achieved throughput. We see that for 100μs interrupt hold off, both Prime and Linux achieve similar throughput of approximately 460Mb/s and 480Mb/s respectively. Prime uses much less CPU achieving the result (32% versus 72%). However, we make no claim of a fair comparison as Linux has a heavier weight IP stack, translation and pinning infrastructure, and uses a socket interface which results in an extra packet copy compared to Prime. We simply observe that we are currently competitive with a traditionally-structured existing system and are optimistic we can at least retain comparable performance in more similarly structured systems. For the 0μs hold-off results, we see Linux goes into livelock near 100% CPU after which throughput tapers off as offered load increases. Prime achieves exactly the same throughput for 0μs and 100μs hold-off, though CPU utilisation differs markedly (58% versus 32%).

6 Conclusions

We have constructed a framework for running device drivers at user-level. Our goal was to preserve the high degree of system integration that enables high-performance driver construction while at the same time confining drivers safely to their own address space like normal applications. We analysed our framework's performance in the context of gigabit ethernet, and our initial results show modest overhead in an execution profile in a test scenario designed to exacerbate the overhead. In throughput oriented benchmarks, we demonstrated similar performance to Linux in terms of achieved throughput. We plan to further explore our framework's performance by constructing more realistic test scenarios (e.g. SPECweb), drivers and interfaces for other devices (e.g. disk). We also plan to explore system structures more comparable to existing systems (e.g. driver, IP stack, and web server all running as separate processes).

We eventually hope that our results will be encouraging enough to CPU and system architects to consider exploring efficient control of DMA for protection purposes in commodity hardware. Such hardware would ensure that device drivers are just normal applications under the complete control of the operating system.

References

1. Liedtke, J.: Toward real microkernels. Communications of the ACM **39** (1996)
2. Hohmuth, M., Tews, H., Stephens, S.G.: Applying source-code verification to a microkernel - the VFiasco project. In: Proc. 10th SIGOPS European Workshop. (2002)
3. Engler, D.R., Kaashoek, M.F., Jr., J.O.: Exokernel: An operating system architecture for application-level resource management. In: 15th Symp. on Operating Systems Principles, Copper Mountain Resort, CO, ACM (1995)

4. Accetta, M., Baron, R., Bolosky, W., Golub, D., Rashid, R., Tevanian, A., Young, M.: MACH: A new kernel foundation for UNIX development. In: Proc. Summer USENIX. (1986)
5. Cheriton, D.R.: The V kernel: A software based for distribution. IEEE Software **1** (1984) 19–42
6. Rozier, M., Abrossimov, V., Armand, F., Boule, I., Gien, M., Guillemont, M., Hermann, F., Kaiser, C., Langlois, S., Leonard, P., Neuhauser, W.: Chorus distributed operating system. Computer Systems **1** (1988)
7. Tanenbaum, A.S., van Renesse, R., van Staveren, H., Sharp, G.J., Mullender, S.J.: Experiences with the amoeba distributed operating system. Communications of the ACM **33** (1990) 46–63
8. Chou, A., Yang, J., Chelf, B., Hallem, S., Engler, D.: An empirical study of operating systems errors. (In: Proc. 18th Symp. on Operating Systems Principles)
9. Microsoft: Driver signing for windows. Available: http://www.microsoft.com/technet/prodtechnol/winxppro/proddocs/code_signing.asp (2002)
10. Gefflaut, A., Jaeger, T., Park, Y., Liedtke, J., Elphinstone, K., Uhlig, V., Tidswell, J., Deller, L., Reuther, L.: The SawMill multiserver approach. In: 9th SIGOPS European Workshop, Kolding, Denmark (2000)
11. Härtig, H., Baumgartl, R., Borriss, M., Hamann, C.J., Hohmuth, M., Mehnert, F., Reuther, L., Schönberg, S., Wolter, J.: DROPS - OS support for distributed multimedia applications. In: Proc. 8th SIGOPS European Workshop, Sintra, Portugal (1998)
12. von Eicken, T., Basu, A., Buch, V., Vogels, W.: U-net: a user-level network interface for parallel and distributed computing. In: Proc. 15th Symp. on Operating Systems Principles, Copper Mountain, Colorado, USA (1995) 40–53
13. Myrinet: Myrinet. Website: www.myrinet.com (2002)
14. Leslie, B., Heiser, G.: Towards untrusted device drivers. Technical Report UNSW-CSE-TR-0303, School Computer Science and Engineering, University of New South Wales, Sydney, 2052, Australia (2003)
15. Felten, E.W., Alpert, R.D., Bilas, A., Blumrich, M.A., Clark, D.W., Damianakis, S.N., Dubnicki, C., Iftode, L., Li, K.: Early experience with message-passing on the SHRIMP multicomputer. In: Proc. 23rd Symp. on Computer Architecture. (1996) 296–307
16. Rawson III, F.L.: An architecture for device drivers executing as user-level tasks. In: USENIX MACH III Symposium. (1993)
17. Liedtke, J., Elphinstone, K., Schönberg, S., Härtig, H., Heiser, G., Islam, N., Jaeger, T.: Achieved IPC performance. In: 6th Workshop on Hot Topics in Operating Systems (HotOS), Chatham, Massachusetts (1997)
18. Aron, M., Liedtke, J., Park, Y., Deller, L., Elphinstone, K., Jaeger, T.: The SawMill framework for virtual memory diversity. In: Australasian Computer Systems Architecture Conference, Gold Coast, Australia, IEEE Computer Society Press (2001)
19. Liedtke, J., Uhlig, V., Elphinstone, K., Jaeger, T., Park, Y.: How to schedule unlimited memory pinning of untrusted processes or provisional ideas about service-neutrality. In: 7th Workshop on Hot Topics in Operating Systems, Rio Rico, Arizona (1999)
20. Unrau, R., Krieger, O.: Efficient sleep/wake-up protocols for user-level IPC. In: International Conference on Parallel Processing. (1998)
21. Ritchie, D., Neufeld, G.: User level ipc and device management in the raven kernel. In: Proc. USENIX Microkernels and Other Kernel Architectures. (1993)
22. Wienand, I., Macpherson, L.: ipbench. Website: http://ipbench.sourceforge.net/ (2002)

A Generation Ahead of Microprocessor: Where Software Can Drive uArchitecture To?

Jesse Z. Fang

Intel

Abstract. The presentation will start with introduction of Microprocessor Technology Labs at Intel to show the Intel's efforts in the uArchitecture and system software research areas. As all of you know, Moore's law successfully leads Intel microprocessor business in decades. Transistor densities will continue to increase. However, device speed scaling will not follow historical trends and leakage power remains an issue. Meanwhile, new usage models and workloads will continue to demand greater performance. We can't run business as usual. We need to develop uArchitecture features more effectively with power constraints. Software in both applications and systems will play more and more important roles in uArchitecture research and microprocessor design. Emerging applications like mining, recognition and synthesis (MRS) will become the next generation dominated applications. Study of the characterizations of these applications is significant for the next generation of microprocessor design. System software has been changing its landscape as well. There are challenges and opportunities for microprocessor designers and researchers to explore new uArchitecture features to meet the needs of the new application and system software. The presentation will show the potential direction of uArchitecture for such emerging applications also. The talk will discuss the emerging paradigms of programming systems and its impact to uArchitecture design by giving couple examples of how compilation technologies enable HW design.

P.-C. Yew and J. Xue (Eds.): ACSAC 2004, LNCS 3189, p. 270, 2004.

A Cost-Effective Supersampling for Full Scene AntiAliasing

Byung-Uck Kim[1], Woo-Chan Park[2], Sung-Bong Yang[1], and Tack-Don Han[1]

[1] Dept. of Computer Science, Yonsei University, Seoul, Korea,
kimbu@yonsei.ac.kr,
[2] Dept. of Internet Engineering, Sejong University, Seoul, Korea

Abstract. We present a graphics hardware system to implement supersampling cost-effectively. Supersampling is the well-known technique to produce high quality images. However, rendering the scene at a higher resolution requires a large amount of memory size and memory bandwidth. Such costs can be alleviated by grouping subpixels into a fragment with a coverage mask which indicates which part of the pixel is covered. However, this may cause color distortion when several objects either overlap or intersect with each other within a pixel. In order to minimize such errors, we introduce an extra buffer, called the *RuF(Recently used Fragment)-buffer*, for storing the footprint of a fragment most recently used in the color manipulation. In our experiments, the proposed system can produce high quality images as good as supersampling with a smaller amount of memory size and memory bandwidth, compared with the conventional supersampling.

Keywords: Antialiasing, Supersampling, Graphics Hardware, Rendering Algorithm

1 Introduction

With growth of user demand for high quality images, the hardware-supported full scene antialiasing (FSAA) has become commonplace in 3D graphics systems. Artifacts due to aliasing are mostly caused by insufficient sampling. To attenuate such aliasing problem, supersampling has been practiced in the high-end graphics system [2] and begins to be adopted by most pc-level graphics accelerator.

In supersampling, 3D objects are rendered at a higher resolution and then are averaged down to the screen resolution [8]. Hence it requires a large amount of memory size and memory bandwidth. For example, $n \times n$ supersampling requires n^2 times bigger both memory size and memory bandwidth than one-point sampling. Some reduced versions of it have been practiced; sparse supersampling [2] that populates sample points sparsely and adaptive supersampling [1] in which the only discontinuity edges are supersampled. In multi-pass approach, the accumulation buffer [4] has been proposed in which one scene is rendered several times and these images are then accumulated, one at a time, into the accumulation buffer. When the accumulation is done, the result is copied back into the

P.-C. Yew and J. Xue (Eds.): ACSAC 2004, LNCS 3189, pp. 271–281, 2004.

frame buffer for viewing. However, it is obvious that rendering the same scene n times takes n times longer than rendering it just once. Both supersampling and the accumulation buffer are well integrated into the Z-buffer (also called *depth buffer*) algorithm that is adopted by most rendering systems for the hidden surface removal. Moreover, Z-buffer algorithm handles correctly interpenetrating objects.

Rather than rendering each subpixel individually, A-buffer approach [3] groups subpixels into a fragment with a coverage mask that indicates which part of the pixel is covered. Such a representation is efficient in reduction of the memory and bandwidth requirements because it shares the common color value instead of having its own color value per subpixel. To apply Carpenter's blending formulation [3] for antialiasing of opaque objects, fragments should be sorted in the fragment list by their depth value. The fragment lists can be implemented by a pointer-based linked list [6] or a pointer-less approach [9]. For reducing noticeable artifacts, correct subpixel visibility calculations are more important that correct antialiasing of subpixels. Therefore, the more concise depth value representation has been practiced in [5].

This paper presents a cost-effective graphics hardware system that renders the supersampled graphics primitives with full scene antialiasing. In our approach, an area-weighted representation of a fragment using a coverage mask is adopted, as in A-buffer, to reduce the memory and bandwidth requirements. This may cause color distortion when several objects either overlap or intersect with each other within a pixel. In order to minimize such errors, we introduce an extra buffer, called the *RuF (Recently used Fragment)-buffer*, for storing the footprint of a fragment most recently used in the color manipulation. In addition, we introduce the new color blending formulation for minimizing color distortion by referencing the footprint of the RuF-buffer. In our simulation, we compared the amount of memory size and memory bandwidth of the proposed scheme with those of supersampling and investigated the per-pixel color difference of the images produced from both methods. For various 3D models with 8 sparse sample points, the proposed algorithm reduces the amount of memory size and memory bandwidth by 35.7% and 67.1%, respectively, with 1.3% per-pixel color difference as compared with supersampling.

The rest of this paper is organized as follows. In Section 2, we describe the proposed graphics architecture. Section 3 explains fragment processing algorithm for antialiasing. Section 4 provides the experimental results of image quality, memory and bandwidth requirement. Finally, the conclusions are given in Section 5.

2 The Proposed Graphics Architecture

In this section, we present the data structure and memory organization for processing a pixel with subpixels individually or a fragment. We also describe the proposed graphics hardware with the newly developed RuF-buffer.

2.1 Data Structure and Memory Organization for a Pixel

Figure 1 shows the data structure and memory organization for representing a pixel with subpixels individually and a fragment. Here we assumed that each pixel has 8 sparse sample points. In supersampling method, each subpixel is processed individually; the pair of color and depth value per subpixel is stored into color buffers and depth buffers in the frame buffer. Hence, the required memory size per pixel is $m \times (C + Z)$ bits where C and Z is color (32 bits) and depth value (24 bits), respectively and m is the number of sample points. In this example, $8 \times (32 + 24)$ bits $=$ 56 bytes per pixel is required.

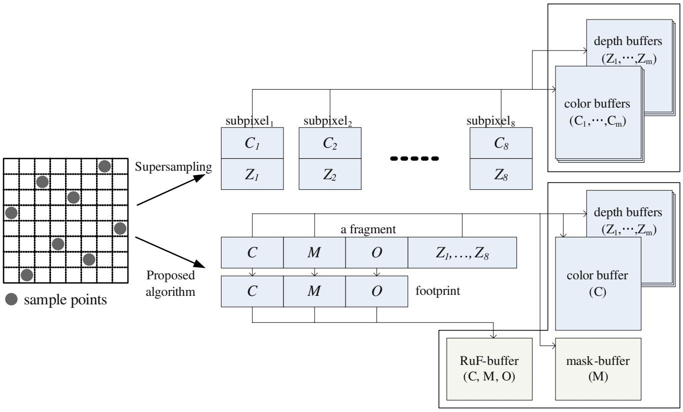


Fig. 1. Data structure and memory organization for a pixel.

In the proposed scheme, the data structure for a fragment is basically originated from the one of the A-buffer. Subpixels within a pixel are grouped into a fragment that shares the common color value (C) with a coverage mask (M). Moreover, we can easily compute the color contribution of a fragment within a pixel since a coverage mask represents an area-weighted value. For handling subpixel visibility correctly, depth value per subpixel $(Z_1, \cdots, Z_m)$ is kept individually. An object tag (O) is the unique identifier per object and can be generated sequentially by the rendering hardware incorporated with modelling software [6]. It is used for post-merging fragments; if two fragments in a pixel have the same object tag value then the footprint of both fragments can be merged into the RuF-buffer. The RuF-buffer holds the footprint of a fragment that is recently used in the color manipulation phase. The footprint of a fragment consists of color, coverage mask and object tag of a fragment and it will be used for correct handling the hidden surface removal. The required memory size per a pixel is $(2 \times (C + M) + m \times Z + O)$ bits where M is the coverage mask (m bits) and O is the object tag (16 bits). Here, 2^{16} objects are assumed to be enough for representing 3D model in a scene. Therefore, the memory size

Table 1. Memory requirement

m	Supersampling	Our approach	Reduction ratio
4	28 bytes	24 bytes	14.3%
8	56 bytes	36 bytes	35.7%
16	112 bytes	62 bytes	44.6%
64	448 bytes	218 bytes	51.3%

of $(2 \times (32 + 8) + 8 \times 24 + 16)$ bits $= 36$ bytes per pixel is required when 8 sparse sample points are used.

Table 1 shows the comparison of the memory requirement between supersampling and the proposed algorithm as the number of sample points increases. As shown in the results, the reduction ratio of the memory requirement begins to be larger as the number of sample points increases since our approach can save the memory requirement for representing individual color value per subpixel by sharing the common color value.

2.2 RuF-Buffer Graphics Architecture

Figure 2 shows the proposed graphics architecture with the conventional geometric-processing and rasterizer-processing. We add the mask-buffer and the RuF-buffer into the conventional architecture. Generally, 3D data are geometric-processed with rotating, scaling and translation. The processed results are fed into the rasterizer-processing. In rasterizer-processing, the fragments of each polygon are generated by scan-conversion and then passed through occlusion test such as Z-buffer algorithm and various image mapping such as texture mapping or bump mapping. Finally, the color value of each pixel is manipulated and stored into the color buffer in the frame buffer. When all the fragments are processed, the color values in the frame buffer are sent to a display device.

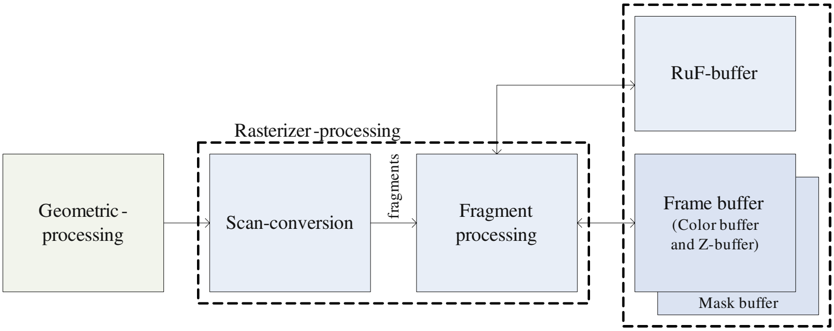


Fig. 2. The proposed graphics architecture.

3 Fragment Processing for Antialiasing

Figure 3 shows three phases of the newly introduced functional unit of fragment processing: the *occlusion test*, the *color manipulation*, and the *RuF-buffer recording*. Roughly speaking, the newly fragment incoming into the graphics pipeline is tested with Z-buffer algorithm per subpixel. If it is totally occluded by the one previously stored in the frame buffer, called a *prepixel*, then it will be discarded and the next fragment will be processing.

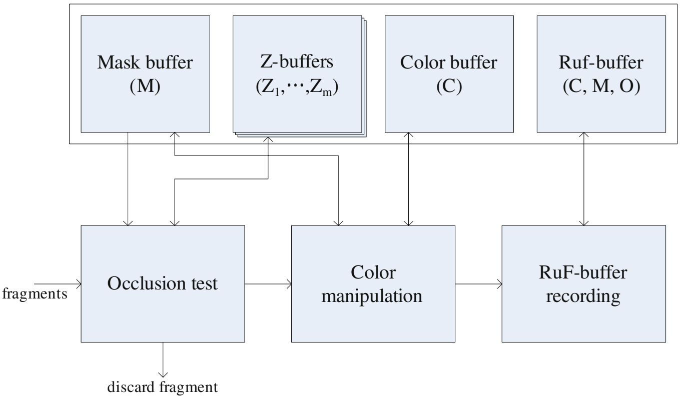


Fig. 3. Functional units for fragment processing.

Otherwise, we calculate the visible fraction of an incoming fragment, called a *survived surface*, and the hidden surface of the prepixel occluded by it. For calculating color value of a pixel, the survived surface will be added into and the hidden surface will be subtracted from the color buffer. In this phase, we look up the RuF-buffer for investigating the color value of the hidden surface. Finally, the survived fragment is merged into the RuF-buffer to allow more opportunity by covering the larger portion within a pixel. In describing each stage, the subscripts, 'i', 'p', and 'r' are used for denoting the attribute of an incoming fragment, the prepixel in the frame buffer, and the footprint in the RuF buffer. For instance, M_i is for the coverage mask of an incoming fragment. For simplicity, we assume that a coverage mask used in formulation returns the area-weighted value; for instance, if the number of sample points is eight and M_i covers three subpixels then M_i in formulation denotes the value of 3/8.

Detail descriptions of each stage in the fragment processing are presented as follows:

The occlusion test : The depth comparison per subpixel between an incoming fragment and a prepixel are tested with the conventional Z-buffer algorithm. Then the mask composite for the survived surface (M_s) and the hidden surface

(M_h) are processed. Z-buffers are updated with new depth values of the survived surface.

The color manipulation : The survived surface is visible fraction of an incoming fragment. Hence, its area-weighted color value should be added into the color buffer. In addition, the hidden surface of a prepixel occluded by the survived surface should be subtracted from the color buffer. To look up the color value of the hidden surface, we investigate the match between the hidden surface and the footprint of the RuF-buffer through the mask comparison of ($M_k = M_h \cap M_r$). If M_k is a subset of M_r then we can totally remove the color contribution of the hidden surface from the frame buffer using the formulation of Eq. 1.

$$\text{new } C_p = C_p + C_i \times M_s - C_r \times M_k \tag{1}$$

However, since the footprint of the RuF-buffer may not provide any information about some parts of the hidden surface we expand the formulation of Eq. 1 to compensate color value with a slight error. The fourth term of formation in Eq. 2 compensates color value by subtracting the area-weighted color value for the blind parts ($M_b = M_h - M_k$) of the hidden surface from the frame buffer.

$$\text{new } C_p = C_p + C_i \times M_s - C_r \times M_k - C_p \times M_b \tag{2}$$

The RuF-buffer recording : Generally, the polygonal surfaces of an object exist in a coplanar space. Therefore, each neighbored surface generates fragments that share the same pixel on their boundary [3],[6]. So, they can be merged into one in the post-processing. Fragments that come from the same object will be merged into one since the same tagged object has same property. The merging process can be computed as follows:

$$\text{new } M_r = M_r \cup M_s ; \text{new } C_r = C_r \times \frac{M_r}{\text{new } M_r} + C_r \times \frac{M_s}{\text{new } M_r} \tag{3}$$

However, if the survived fragment has the different object tag then the RuF-buffer is reset with the survived surface. The new object now begins to be drawn.

Figure 4 and Table 2 shows an example of fragment processing for each event and its associated color manipulation. In this example, subpixels are located on 3×3 grid sample points and three consecutive fragments (f_1, f_2, f_3) are incoming into the graphics pipeline. In Figure 4, we assume that a fragment f_1 was already processed in the previous phase; the frame buffer was initialized and then filled with f_1, where f_1 of object one (O_1) covers four subpixels with a color value C_0. Hence, the color value of a prepixel in the frame buffer (C_1) was computed as an area-weighted value of f_1, and then the footprint of f_1 was stored in the RuF-buffer. Now two fragments, f_2 and f_3, are newly fed into the graphics pipeline sequentially.

The left on the figure shows the processing of a fragment f_2 which of object 2 (O_2) has C_2 as a color value and covers four subpixels. The occlusion test is

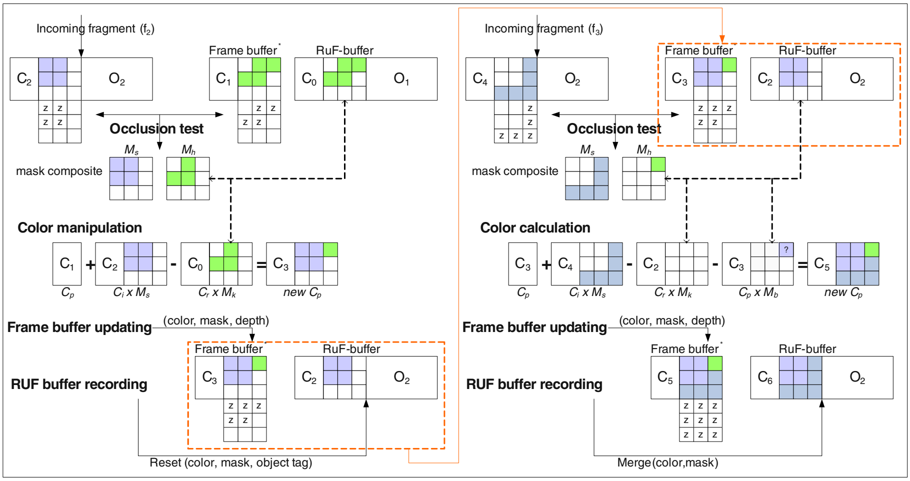


Fig. 4. Example of fragment processing

Table 2. The color manipulation process

Events	Color manipulation	Color buffer
Frame buffer initialize	–	$\emptyset$
f_1 incoming	$C_0 \times \frac{4}{9}$	C_1
f_2 incoming	$C_1 + C_2 \times \frac{4}{9} - C_0 \times \frac{3}{9}$	C_3
f_3 incoming	$C_3 + C_4 \times \frac{5}{9} - C_3 \times \frac{1}{9}$(instead of $C_0 \times \frac{1}{9}$)	C_5

first processed, and then the survived and hidden masks are composited by Z-buffer algorithm per subpixel. In this example, four subpixels of f_2 are survived and three subpixels in a prepixel are occluded. Moreover, the information of the hidden surface can be referenced through the RuF-buffer. Thus the new color value (C_3) can be computed by Eq. 1 without any color distortion. Finally, the RuF-buffer is reset with the footprint of f_2 since the object tag of f_2 is different to the one of the RuF buffer stored in the previous phase.

The right on the figure shows the processing of a fragment f_3, which of object 2 (O_2) has C_4 as a color value and covers five subpixels. Similarly as in f_2 processing, the hidden and survived surfaces are computed in the occlusion test; in this example, five subpixels are survived and one subpixel is occluded. However, in the color manipulation, the footprint of the RuF-buffer cannot provide any information of the hidden surface (M_b). So, the color value is compensated by subtracting it from the prepixel; for instance, the color value of $C_3 \times \frac{1}{9}$ are used in formulation instead of $C_0 \times \frac{1}{9}$. This causes color distortion with the mere color difference of ($C_3 \times \frac{1}{9} - C_0 \times \frac{1}{9}$). In RuF-buffer recording, the footprint of two fragments f_2 and f_3 are merged into one since they have the same object tag, and then it covers the entire portion of a pixel.

4 Empirical Results

In our experiments, the 3D models described with OpenGL functions are geometric-processed and passed through scan-conversion in the Mesa library, which is the OpenGL-clone implementation and can be accessed in public domain [7]. We modified the Mesa library to output the tracefile of a fragment with a coverage mask. The resulted tracefile is fed into the simulator that implements the proposed architecture in C. Then the final image of 200 × 150 resolution is produced as shown in Figure 5.

Table 3 describes the characteristics of 3D models used in our experiments where the number of vertices (V), triangles (T), fragments (F), and objects (O) are provided. In our experiment, we decided to use eight-sparse sample point ($8 \times RuF$) for the antialiasing architecture because it has been successfully practiced in high-end graphics systems [2]. To provide an indication of the performance in our approach, various supersampling methods are also simulated; one-point sampling ($1 \times S$, 1 subpixel per pixel), 8 sparse supersampling ($8 \times S$, 8 subpixels per pixel), 4 by 4 supersampling ($4 \times 4S$, 16 subpixels per pixel), and 8 by 8 supersampling ($8 \times 8S$, 64 subpixels per pixel).

Table 3. The characteristics of 3D models used in our experiments

Model Name	V	T	F	O
Al	3618	7124	11975	35
Castle	6620	13114	17444	16
Dolphins	885	1692	4570	3
Pig	3522	7040	7499	3
Rose+vase	4028	3360	5425	5
Teapot	3644	6320	6807	1
Venus	711	1418	5464	1

4.1 Image Quality

We present the image quality with respect to the number of sample points. To observe the quality of final scenes, the error metric of per-pixel color difference is used as shown in Eq. 4.

$$\text{Per-pixel color difference} = \sum_{\forall i,j} \sum_{c=r,g,b} (p_{ijc} - q_{ijc})^2, \quad (4)$$

where p_{ij} and q_{ij} are the pixels from the same location of a reference image and a test image, respectively.

In order to make a small number of pixels with large difference more noticeable, the square of the difference is made [5]. We compute the per-pixel color difference for each 3D model where the reference image is produced by $8 \times 8S$,

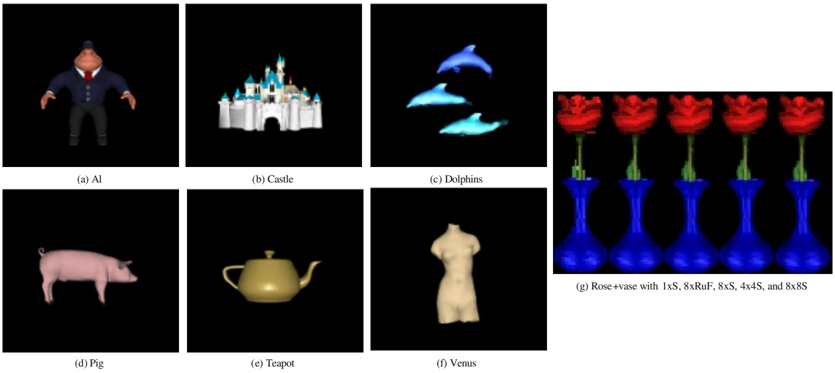

(a) Al (b) Castle (c) Dolphins (d) Pig (e) Teapot (f) Venus (g) Rose+vase with 1xS, 8xRuF, 8xS, 4x4S, and 8x8S

Fig. 5. The final images for each 3D model

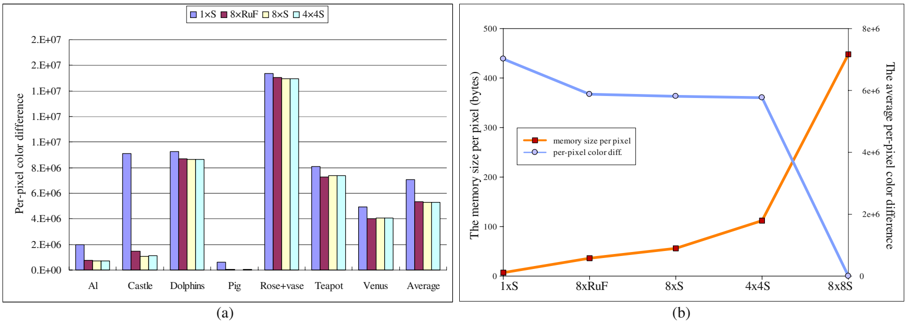


Fig. 6. Performance of color difference and memory size.

it is regarded as to be an ideal image, and each test image is produced by $1 \times S$, $8 \times S$, $8 \times RuF$ and $4 \times 4S$, respectively.

Figure 6(a) shows the results of the per-pixel color difference for each 3D dataset with various sample points. As can be seen from the results, the per-pixel color difference becomes to be smaller as the number of sampling points increase. Moreover, the final images of $8 \times RuF$ are almost as good quality as $8 \times S$; both of them have the same number of sample points.

4.2 Trade-Off Between Image Quality and Memory Requirement

In order to show the cost efficiency of the proposed architecture, two graphs for memory size per pixel and per-pixel color difference, respectively, are plotted together in Figure 6(b). The per-pixel color difference between $8 \times RuF$ and $8 \times S$ is 1.3% but the memory size per pixel is 35.7%. That is, our approach provides almost as good quality as supersampling with a less hardware cost.

4.3 Memory Bandwidth Requirement

Figure 7 shows the memory bandwidth requirements where two bar graphs for supersampling (left) and the proposed scheme (right) are plotted as a pair for each model. Arbitrary scenes for each 3D model are produced with both methods where 8 sparse sample points are used. As shown in the results, the proposed architecture can reduce the memory bandwidth requirement by 53.6% ∼ 75.5% for Castle and Rose+vase. The internal bandwidth is required for pixel processing between the graphics pipeline and the frame buffer (includes the RuF-buffer and mask-buffer). The external bandwidth is for swapping the front and back buffer or for average-down filtering. In supersampling, the external bandwidth dominates the memory bandwidth requirement. In other words, it implies that the screen-size color buffer is very efficient in reducing the bandwidth requirement since it does not require the overhead for average-down filtering process.

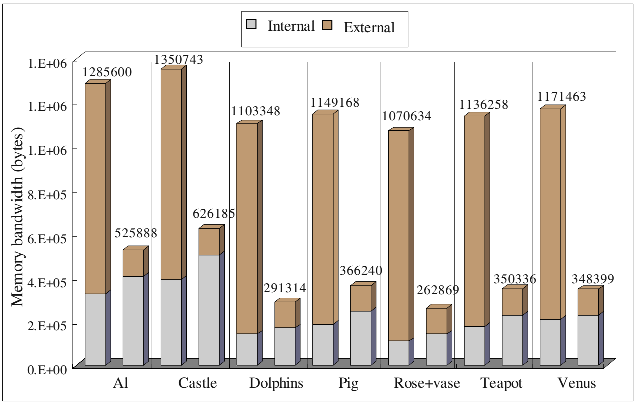


Fig. 7. The memory bandwidth requirement

5 Conclusion

In this paper, we present a graphics hardware system to implement supersampling in cost-effective manner. For hardware-implementation aspect, our graphics architecture uses same programming model as in Z-buffer algorithm for the hidden surface removal and adds only small additions to the conventional rendering process such as mask comparison and composite. In addition, mask comparison and composite can be simply processed with bitwise operations. In the color manipulation, computing the color contribution of a fragment can be processed through look-up tables, each entry of which holds the predefined floating point number divided by the number of sample points.

To provide an indication of the performance in terms of cost-effective full scene antialiasing, the results of memory requirement, bandwidth requirement,

Table 4. Summary of performance

	Memory size	Memory bandwidth	per-pixel color diff.
$8 \times S$	56 Bytes	1181030 Bytes	1338250
$8 \times RuF$	35 Bytes	395890 Bytes	1336154
Reduction ratio	35.7%	67.1%	1.3% (difference)

and per-pixel color difference are shown in Table 4 when 8 sparse sample points are used. It shows that the proposed architecture can reduce the memory size and the memory bandwidth size by 35.7% and by 67.1% with a slight color difference of 1.3%, compared with the conventional supersampling. As shown in the results, the proposed architecture can efficiently render the high quality scene with an economic hardware cost. Moreover, the simplicity of rendering process for our scheme allows us to have fast rendering through well-defined pipeline with a single pass.

Acknowledgment. The authors are grateful to the anonymous reviewers of the earlier version of this paper, whose incisive comments helped improve the presentation. This work is supported by the NRL-Fund from the Ministry of Science & Technology of Korea.

References

1. Aila, F., Miettine, V., Nord, P.: Delay Streams for Graphics Hardware. ACM Transactions on Graphics **22** (2003) 792–800
2. Akeley, K.: RealityEngine graphics. Computer Graphics (SIGGRAPH 93) **27** (1993) 109–116
3. Carpenter, L.: The A-buffer: an Antialiased Hidden Surface Method. Computer Graphics (SIGGRAPH 84) **18** (1984) 103–108
4. Haeberli, P.E., Akeley, K.: The Accumulation Buffer: Hardware Support for High-Quality Rendering. Computer Graphics (SIGGRAPH 90) **24** (1990) 309–318
5. Jouppi, N.P., Chang, C.F.: Z^3: an Economical Hardware Technique for High-quality Antialiasing and Transparency. In Proceeding of Graphics hardware (1993) 85–93
6. Lee, J.A., Kim, L.S.: Single-Pass Full-Screen Hardware Accelerated Antialiasing. In Proceeding of Graphics hardware (2000) 67–75
7. The Mesa 3D Graphics Library. `http://www.mesa3d.org`
8. Watt, A.: 3D Computer Graphics. Third Edition (2000) Addison-Wesley
9. Wittenbrink, C.M.: R-Buffer: A Pointless A-Buffer Hardware Architecture. In Proceeding of Graphics hardware (2001) 73–80

A Simple Architectural Enhancement for Fast and Flexible Elliptic Curve Cryptography over Binary Finite Fields GF(2^m)

Stefan Tillich and Johann Großschädl

Graz University of Technology
Institute for Applied Information Processing and Communications
Inffeldgasse 16a, A–8010 Graz, Austria
{Stefan.Tillich,Johann.Groszschaedl}@iaik.at

Abstract. Mobile and wireless devices like cell phones and network-enhanced PDAs have become increasingly popular in recent years. The security of data transmitted via these devices is a topic of growing importance and methods of public-key cryptography are able to satisfy this need. Elliptic curve cryptography (ECC) is especially attractive for devices which have restrictions in terms of computing power and energy supply. The efficiency of ECC implementations is highly dependent on the performance of arithmetic operations in the underlying finite field. This work presents a simple architectural enhancement to a general-purpose processor core which facilitates arithmetic operations in binary finite fields GF(2^m). A custom instruction for a multiply step for binary polynomials has been integrated into a SPARC V8 core, which subsequently served to compare the merits of the enhancement for two different ECC implementations. One was tailored to the use of GF(2^{191}) with a fixed reduction polynomial. The tailored implementation was sped up by 90% and its code size was reduced. The second implementation worked for arbitrary binary fields with a range of reduction polynomials. The flexible implementation was accelerated by a factor of nearly 10.

Keywords: Elliptic curve cryptography, application-specific instruction set extension, binary finite fields, SPARC V8, multiply step instruction.

1 Introduction

Security for mobile and wireless applications requires the involved devices to perform cryptographic operations. For open systems, the use of public-key cryptography is practically inevitable. There are likely to be two groups of devices which will participate in secure mobile and wireless environments [17]: end devices and servers. End devices are often constrained regarding computing power, memory for software code, RAM size and energy supply. Those devices require fast, memory- and energy-efficient implementations of public-key methods. Elliptic curve cryptography (ECC) reduces the size of the operands involved in computation (typically 160–250 bit) compared to the widely used RSA cryptosystem (typically 1024–3072 bit) and is therefore an attractive way to realize

P.-C. Yew and J. Xue (Eds.): ACSAC 2004, LNCS 3189, pp. 282–295, 2004.

security on constrained devices. With typical processor word-sizes of 8–64 bit, public-key cryptosystems call for efficient techniques to handle multiple-precision operands. Binary finite extension fields GF(2^m) allow efficient representation and computation on a general-purpose processor which does not feature a hardware multiplier and are therefore well suited to be used as the underlying field of an elliptic curve cryptosystem.

ECC implementations require several choices of parameters regarding the underlying finite field (type of the field, representation of its elements, and the algorithms for the arithmetic operations) as well as the elliptic curve (representation of points, algorithms for point arithmetic). If some of these parameters are fixed, e.g. the field type, then implementations can be optimized yielding a considerable performance gain. Such an optimized ECC implementation will mainly be required by constrained end devices in order to cope with their limited computing power. The National Institute of Standards and Technology (NIST) has issued recommendations for specific sets of parameters [13]. As research in ECC advances, new sets of parameters with favorable properties are likely to become available and recommended. Therefore, not all end devices will use the same set of parameters. Server machines which must communicate with many different clients will therefore have a need for flexible and yet fast ECC implementations.

This paper introduces a simple extension to a general-purpose processor to accelerate the arithmetic operations in binary extension fields GF(2^m). Our approach concentrates on a very important building block of these arithmetic operations; namely the multiplication of binary polynomials, i.e. polynomials with coefficients in GF$(2) = \{0, 1\}$. If this binary polynomial multiplication can be realized efficiently, then multiplication, squaring and inversion in GF(2^m) and in turn the whole ECC operation is made significantly faster.

Two forms of a multiply step instruction are proposed, which can be implemented and used separately or in combination. These instructions perform an incremental multiplication of two binary polynomials by processing one or two bit(s) of one polynomial and accumulating the partial products. A modified ripple-carry adder is presented which facilitates the accumulation with little additional hardware cost. The proposed custom instructions have merits for implementations which are optimized for specific binary finite fields with a fixed reduction polynomial. Also, flexible implementations which can accommodate fields of arbitrary length with a range of reduction polynomials benefit from such instructions. Both types of implementations are general enough to support different elliptic curves and EC point operation algorithms.

The remainder of this paper is organized as follows. Some principles of elliptic curve cryptography in binary finite fields are given in the next Section. Section 3 outlines important aspects of modular multiplication in GF(2^m). A modified ripple-carry adder which facilitates the implementation of our enhancement is presented in Section 4. Section 5 describes the proposed custom instructions in detail and Section 6 gives evaluation results from our implementation on an FPGA-board. Finally, conclusions are drawn in Section 7.

2 Elliptic Curve Cryptography

An elliptic curve over a field $\mathbb{K}$ can be formally defined as the set of all solutions $(x, y) \in \mathbb{K} \times \mathbb{K}$ to the general (affine) Weierstraß equation

$$y^2 + a_1xy + a_3y \;=\; x^3 + a_2x^2 + a_4x + a_6 \qquad (1)$$

with the coefficients $a_i \in \mathbb{K}$. If $\mathbb{K}$ is a *finite field* GF(q), then the set of all pairs (x, y) satisfying Equation (1) is also finite. A finite field GF(q) is also called a *Galois field*. If the finite field is a binary extension field GF(2^m), then Equation (1) can be simplified to

$$y^2 + xy \;=\; x^3 + ax^2 + b \quad \text{with} \quad a, b \in \mathrm{GF}(2^m) \qquad (2)$$

The set of all solutions $(x, y) \in \mathrm{GF}(2^m) \times \mathrm{GF}(2^m)$, together with an additional special point $\mathcal{O}$, which is called the "point at infinity", forms an Abelian group whose identity element is $\mathcal{O}$. The group operation is the addition of points, which can be realized with addition, multiplication, squaring and inversion in GF(2^m). A variety of algorithms for point addition exists, where each requires a different number of those field operations. If, e.g. the points on the elliptic curve are represented in *projective coordinates* [2], then the number of field inversions is reduced at the expense of additional field multiplications.

All EC cryptosystems are based on an computation of the form $Q = k \cdot P$, with P and Q being points on the elliptic curve and $k \in \mathbb{N}$. This operation is called *point multiplication* (or *scalar multiplication*) and is defined as adding P exactly $k - 1$ times to itself: $k \cdot P = P + P + \cdots + P$. The execution time of the scalar multiplication is crucial to the overall performance of EC cryptosystems. Scalar multiplication in an additive group corresponds to exponentiation in a multiplicative group. The inverse operation, i.e. to recover k given P and $Q = k \cdot P$, is denoted as the elliptic curve discrete logarithm problem (ECDLP), for which no subexponential-time algorithm has been discovered yet. More information on EC cryptography is available from various sources, e.g. [2,8].

3 Arithmetic in Binary Extension Fields GF(2^m)

A common representation for the elements of a binary extension field GF(2^m) is the polynomial basis representation. Each element of GF(2^m) can be expressed as a binary polynomial of degree at most $m - 1$.

$$a(t) \;=\; \sum_{i=0}^{m-1} a_i \cdot t^i \;=\; a_{m-1} \cdot t^{m-1} + \cdots + a_1 \cdot t + a_0 \quad \text{with} \quad a_i \in \{0, 1\} \qquad (3)$$

A very convenient property of binary extension fields is that the addition of two elements is done with a simple bitwise XOR, which means that the addition hardware does not need to deal with carry propagation in contrast to a conventional adder for integers. The instruction set of virtually any general-purpose processor includes an instruction for the bitwise XOR operation.

Algorithm 1. Multiple-precision multiplication of binary polynomials [5]

```
Input: Two binary polynomials, $a(t) = (\tilde{a}_{s-1}, \ldots, \tilde{a}_1, \tilde{a}_0)$ and $b(t) = (\tilde{b}_{s-1}, \ldots, \tilde{b}_1, \tilde{b}_0)$,
    each represented by an array of $s$ single-precision (i.e. $w$-bit) words.
Output: Product $r(t) = a(t) \cdot b(t) = (\tilde{r}_{2s-1}, \ldots, \tilde{r}_1, \tilde{r}_0)$.
$(\tilde{u}, \tilde{v}) \leftarrow 0$
for $i$ from 0 by 1 to $s-1$ do
    for $j$ from 0 by 1 to $i$ do
        $(\tilde{u}, \tilde{v}) \leftarrow (\tilde{u}, \tilde{v}) \oplus (\tilde{a}_j \otimes \tilde{b}_{i-j})$
    end for
    $\tilde{r}_i \leftarrow \tilde{v}$
    $\tilde{v} \leftarrow \tilde{u}$ , $\tilde{u} \leftarrow 0$
end for
for $i$ from $s$ by 1 to $2s-2$ do
    for $j$ from $i-s+1$ by 1 to $s-1$ do
        $(\tilde{u}, \tilde{v}) \leftarrow (\tilde{u}, \tilde{v}) \oplus (\tilde{a}_j \otimes \tilde{b}_{i-j})$
    end for
    $\tilde{r}_i \leftarrow \tilde{v}$
    $\tilde{v} \leftarrow \tilde{u}$ , $\tilde{u} \leftarrow 0$
end for
$\tilde{r}_{2s-1} \leftarrow \tilde{v}$
return $r(t) = (\tilde{r}_{2s-1}, \ldots, \tilde{r}_1, \tilde{r}_0)$
```

When using a polynomial basis representation, the multiplication in GF(2^m) is performed modulo an *irreducible polynomial* $p(t)$ of degree exactly m. In general, a multiplication in GF(2^m) consists of multiplying two binary polynomials of degree up to $m-1$, resulting in a product-polynomial of degree up to $2m-2$, and then reducing this product modulo the irreducible polynomial $p(t)$ in order to get the final result. The simplest way to implement the multiplication of two binary polynomials $a(t), b(t) \in$ GF(2^m) in software is by means of the so-called *shift-and-xor method* [14]. In recent years, several improvements of the classical shift-and-xor method have been proposed [7]; the most efficient of these is the *left-to-right comb method* by López and Dahab [11], which employs a look-up table to reduce the number of both shift and XOR operations.

A completely different way to realize the multiplication of binary polynomials in software is based on the MULGF2 operation as proposed by Koç and Acar [9]. The MULGF2 operation performs a word-level multiplication of binary polynomials, similar to the $(w \times w)$-bit MUL operation for integers, whereby w denotes the word-size of the processor. More precisely, the MULGF2 operation takes two w-bit words as input, performs a multiplication over GF(2) treating the words as binary polynomials, and returns a $2w$-bit word as result. All standard algorithms for multiple-precision arithmetic of integers can be applied to binary polynomials as well, using the MULGF2 operation as a subroutine [5]. Unfortunately, most general-purpose processors do not support the MULGF2 operation in hardware, although a dedicated instruction for this operation is simple to implement [12]. It was shown by the second author of this paper [6] that a conventional integer multiplier can be easily extended to support the MULGF2 operation, without significantly increasing the overall hardware cost. On the other hand, Koç and Acar [9] describe two efficient techniques to "emulate" the MULGF2 operation when it is not supported by the processor. For small word-sizes (e.g. $w = 8$), the MULGF2 operation can be accomplished with help of look-up tables. The second approach is to emulate MULGF2 using shift and XOR operations (see [9] for further details).

In the following, we briefly describe an efficient word-level algorithm for multiple-precision multiplication of binary polynomials with help of the MULGF2 operation. We write any binary polynomial $a(t) \in \mathrm{GF}(2^m)$ as a bit-string of its m coefficients, e.g. $a(t) = (a_{m-1}, \ldots, a_1, a_0)$. Then, we split the bit-string into $s = \lceil m/w \rceil$ words of w bits each, whereby w is the word-size of the target processor. These words are denoted as $\tilde{a}_i$ (for $0 \leq i < s$), with $\tilde{a}_{s-1}$ and $\tilde{a}_0$ representing the most and least significant word of $a(t)$, respectively. In this way, we can conveniently store a binary polynomial $a(t)$ in an array of s single-precision words (unsigned integers), i.e. $a(t) = (\tilde{a}_{s-1}, \ldots, \tilde{a}_1, \tilde{a}_0)$. Based on the MULGF2 operation, a multiple-precision multiplication of binary polynomials can be performed according to Algorithm 1, which is taken from a previous paper of the second author [5]. The tuple $(\tilde{u}, \tilde{v})$ represents a double-precision quantity of the form $u(t) \cdot t^w + v(t)$, i.e. a polynomial of degree $2w - 1$. The characters $\otimes$ and $\oplus$ denote the MULGF2 and XOR operation, respectively. In summary, Algorithm 1 requires to carry out s^2 MULGF2 operations and $2s^2$ XOR operations in order to calculate the product of two s-word polynomials. We refer to the original paper [5] for a detailed treatment of this algorithm.

Once the product $a(t) \cdot b(t)$ has been formed, it must be reduced modulo the irreducible polynomial $p(t) = t^m + \sum_{i=0}^{m-1} p_i \cdot t^i$ to obtain the final result (i.e. a binary polynomial of degree up to $m - 1$). This reduction can be implemented very efficiently when $p(t)$ is a sparse polynomial, which means that $p(t)$ has few non-zero coefficients c_i. In such case, the modular reduction requires only a few shift and XOR operations and can be highly optimized for a given irreducible polynomial [14,7,8]. Most standards for ECC, such as from ANSI [1] and NIST [13], propose to use sparse irreducible polynomial like trinomials or pentanomials. On the other hand, an efficient word-level reduction method using the MULGF2 operation was introduced in the previously mentioned paper [5]. The word-level method also works with irreducible polynomials other than trinomials or pentanomials, but requires that all non-zero coefficients (except of p_m) are located within the least significant word of $p(t)$, i.e. $p_i = 0$ for $w \leq i < m$. For example, we used the trinomial $t^{191} + t^9 + 1$ for our ECC implementations, which satisfies this condition for a word-size of $w = 32$.

4 Modified Ripple-Carry Adder

A previous paper of the second author [6] presents the design of a so-called unified multiply-accumulate unit that supports the MULGF2 operation. The efficiency of that design is based on integration of polynomial multiplication into the datapath of the integer multiplier. On the other hand, the datapath for our proposed multiply step instructions can be integrated into the ALU adder and does not require a multiplier. For SPARC V8 cores, the implementation of our extension is relatively easy, as those cores already feature a multiply step instruction for integer arithmetic. In comparison to the previous work [6], the multiply step instructions offer a tradeoff of hardware cost against speed.

The simplest way to implement adders in general-purpose processors is in the form of ripple-carry adders. For instance the SPARC V8 LEON-2 processor, which we have used for our evaluation, employs a such an adder. Principally, ripple-carry adders consist of a chain of full adder cells, where each cell takes three input bits (usually labeled a, b and c_{in}) and produces two output bits with different significance (sum and c_{out}). The cells are connected via their carry signals, with the c_{out} of one stage serving as c_{in} input for the next higher stage.

A conventional ripple-carry adder takes two n-bit values and a carry-in bit and produces a n-bit sum and a carry-out bit which can be seen as the $(n+1)$-th bit of the sum. To generate a bit of the *sum* vector, each full adder cell performs a logical XOR of its three inputs a, b and c_{in}. This property can be exploited to perform a bitwise logical XOR of three n-bit vectors with a slightly modified ripple-carry adder. As explained in Section 3, this XOR conforms to an addition of the three vectors if they are interpreted as binary polynomials.

The modification consists of the insertion of multiplexors into the carry-chain of the ripple-carry adder as illustrated in Figure 1. The *insert* control signal selects the carry value which is used. If *insert* is 0, the adder propagates the carry signal, selecting cp_i as c_{i+1}. In this mode the adder performs a conventional integer addition, setting s and c_{out} accordingly. If *insert* is 1, the carry is not propagated, but the *cins* vector is used to provide the c_i inputs for the full adder cells. The *sum* vector s is calculated as the bitwise logical XOR of the vectors a, b and *cins*. The value of c_{out} is not relevant in this mode. In Figure 1 the bits with the same significance of the three vectors are grouped together by braces. The carry input of the rightmost full adder cell acts as c_{in} for integer addition and as least significant bit of the *cins* vector for addition of binary polynomials. The *insert* signal of the modified adder therefore switches between the functionality of an integer adder and and a 3:1 compressor for binary polynomials.

Ripple-carry adders have the disadvantage that the delay of carry propagation can be rather high. Embedded processors normally feature other, longer combinational paths, so that the carry propagation delay is not the critical path delay. If however the carry propagation path of the adder constitutes the critical path and the proposed modifications increase its delay significantly, other approaches are possible to get the 3:1 compressor functionality for binary polynomials. One solution is to modify a faster adder, e.g. a carry-select adder [3]. Another possibility is the use of dedicated XOR-gates without any modification of the adder. Both of these options come with an increased hardware cost.

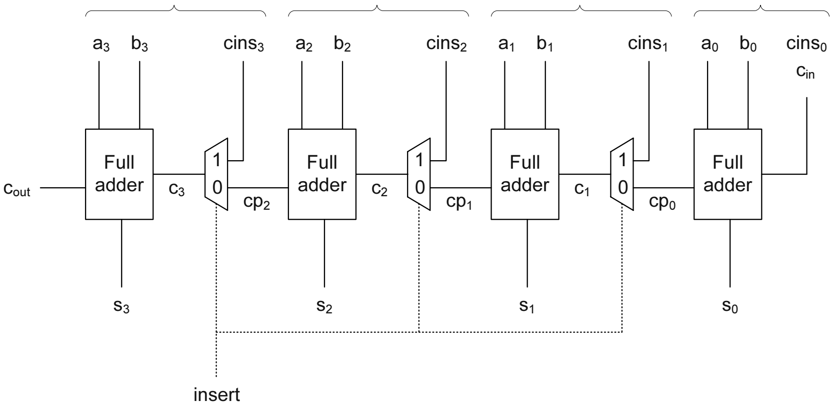


Fig. 1. A 4-bit modified ripple-carry adder

5 Multiply Step Instruction

Our enhancement is basically the addition of one or two custom instructions which can be realized with relatively little additional hardware. The basic idea is to provide a multiply step instruction for multiplication of binary polynomials. With a given word size w of the processor, a multiplication of two w-bit binary polynomials yielding a $2w$-bit result can be implemented efficiently with the proposed instructions. This word-size multiplication of binary polynomials corresponds to the MULGF2 operation mentioned in Section 3, which is an important building block for arithmetic operations in the field GF(2^m).

The SPARC V8 Architecture Manual [16] defines a multiply step instruction for integer multiplication (MULSCC) and our proposed instructions are a modification thereof. MULSCC processes one bit of the multiplier and adds the resulting partial product to an 64-bit accumulator realized by two hardware registers. In the following, the register naming conventions of SPARC V8 will be used.

In order to perform a complete multiplication of two 32-bit binary polynomials, three registers have to be employed. Two of those registers form a 64-bit accumulator to hold the intermediate total during multiplication. These registers will be named %o0 and %y, with %o0 holding the 32 most significant and %y holding the 32 least significant bits. The register %o1 will be used as the third register. It contains the multiplicand during the whole course of the multiplication.

5.1 MULGFS Instruction

The first proposed instruction is named MULGFS and is only a slight variation of the MULSCC instruction. It can be used in the following fashion the perform a word-size polynomial multiplication (MULGF2 operation): At first, the multiplicand is loaded into %o1 and the multiplier is loaded into %y. Then the MULGFS instruction is executed 32 times to process each bit of the multiplier in %y. In each execution of the `MULGFS` instruction the value in the accumulator (`%o0` and `%y`) is shifted right by one. The bit which is shifted out of the accumulator, i.e. the least significant unprocessed bit of the multiplier, is examined and a partial product is generated: If the bit is one, it is the value of the multiplicand, otherwise it is all zero. This partial product is added to the 32 highest bits of the accumulator, which reside in `%o0`. After 32 `MULGFS` instructions, the value in the accumulator must be shifted right by one to obtain the correct 64-bit result. Following the SPARC conventions, the `MULGFS` instruction is written in the following form in assembly code:

```
MULGFS %o0, %o1, %o0
```

The first two registers are the source registers. The first one (`%o0`) contains the highest 32 bits of the accumulator while the second one (`%o1`) holds the multiplicand. The third register is the destination register (`%o0`) which is normally chosen to be the same as the first source register. The register for the 32 lowest bits of the accumulator (`%y`) is read and written implicitly for multiplication instructions in the SPARC architecture. In this case the 64-bit accumulator is formed by `%o0` and `%y`. On other architectures, different approaches may be favorable, e.g. on a MIPS architecture the multiplication registers `%hi` and `%lo` could be implicitly used as accumulator. In detail, a single `MULGFS` instruction performs the following steps:

1. The value in the first source register (`%o0`) is shifted right by one. The shifted value is denoted as C.
2. The least significant unprocessed bit of the multiplier (last bit of `%y`) is examined. The partial product (denoted as A) is set to the value of the multiplicand (`%o1`), if the bit is one. Otherwise A is set to all zeros.
3. The contents of the `%y` register is shifted right by one with the least significant bit of `%o0` shifted in from the left. The bit of the multiplier, which has been processed in the previous step, is therefore shifted out of `%y`.
4. A bitwise XOR of A and C is performed and the result is stored in the higher word of the accumulator (`%o0`).

The `MULGFS` instruction does not require the insertion of a carry vector for the adder. It is sufficient if the adder can suppress carry propagation whenever a specific control signal is set. The changes to the processor for the implementation of the `MULGFS` instruction are:

- Modifications to the decode logic to recognize the new opcode, and to generate an *insert* control signal for the ALU.
- Multiplexors to select the two operands for the adder and which allow shifting of the value in the two registers which form the accumulator.
- Gates which prevent carry propagation in the adder if *insert* is set.

5.2 MULGFS2 Instruction

The second proposed instruction (named MULGFS2) is a variation of the MULGFS instruction, which processes two bits of the multiplier simultaneously. In this fashion two partial products are generated and addition to the accumulated result can be done with a modified ripple-carry adder as specified as in Section 4.

A multiplication of two binary polynomials (MULGF2 operation) is done in the same way as described in the previous Section with the exception that the 32 subsequent MULGFS instructions are replaced by 16 MULGFS2 instructions. If only the MULGFS2 instruction is available, the final shift of the accumulator must be done with conventional bit-test and shift instructions. On the SPARC V8 this requires four instructions. If the MULGFS instruction is available, then the final shift can be done with a single instruction. The format for the MULGFS2 instruction remains the same as for the MULGFS instruction:

```
MULGFS2 %o0, %o1, %o0
```

In detail, the MULGFS2 instruction works by executing these steps:

1. The value in the first source register (%o0) is shifted right by two. The shifted value is denoted as C.
2. The least significant unprocessed bit of the multiplier (last bit of %y) is examined. If the bit is one, a partial product (denoted as B) is set to the value of the multiplicand (%o1) shifted right by one. Otherwise B is all zeros.
3. The second lowest bit of the multiplier (penultimate bit of %y) is examined. If is is one, the second partial product (denoted as A) is set to the value of the multiplicand (%o1). Otherwise A is all zeros.
4. The contents of the %y register is shifted right by two with the following bits set as the new MSBs: The one but highest bit is set to the value of the least significant bit of %o0. The highest bit results from an XOR of the second lowest bit of %o0 and the logical and of the least significant bit of the multiplicand (%o1) and the second lowest bit of %y.
5. A bitwise XOR of A, B and C is performed and the result is stored in in the higher word of the accumulator (%o0).

The MULGFS2 instruction performs the XOR of the three 32-bit vectors with a modified ripple-carry adder. The required modifications to the processor are:

- Changed decode logic to recognize MULGFS2 instructions, and to generate an *insert* control signal for the modified ripple-carry adder.
- Multiplexors to select the three operands for the adder and which allow shifting by two of the values in the two registers which form the accumulator.
- A modified ripple-carry adder as described in Section 4 which is controlled by the *insert* signal.

The implementation of the MULGFS2 instruction for a SPARC V8 general-purpose processor can be seen in Figure 2.

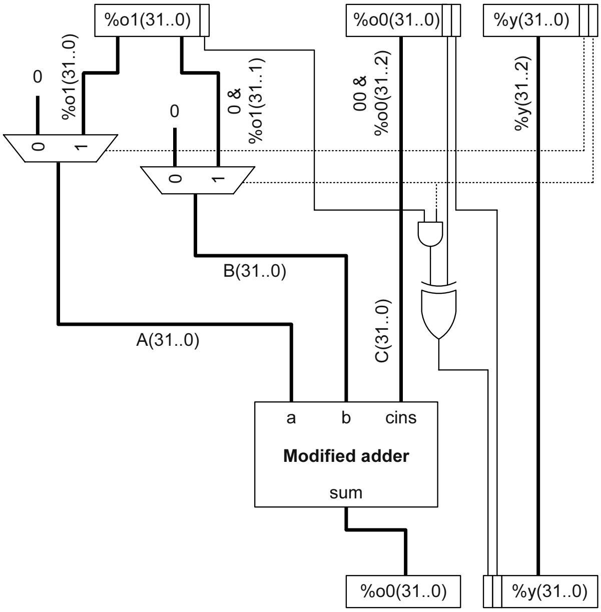


Fig. 2. MULGFS2 instruction implementation for a SPARC V8 processor

6 Experimental Results

Both MULGFS and MULGFS2 instructions have been implemented in the freely available SPARC V8-compliant LEON-2 processor [4]. The size for both instruction and data cache have been set to 4 kB. A tick counter register, whose content is incremented each clock cycle, has also been added to the LEON-2 to facilitate the measurement of the execution time of software routines. A XSV-800 Virtex FPGA prototyping board [18] has been used to implement the extended processor for verification of the design and for obtaining timing result for different realizations of ECC operations.

The ECC parameters given in Appendix J.2.1 of the ANSI standard X9.62 [1] have been used. The elliptic curve is defined over the binary finite field GF(2^{191}) with the reduction polynomial $t^{191} + t^9 + 1$. Most of the examined implementation variants use a multiplication of two binary polynomials (MULGF2 operation) as a building block for GF(2^m) operations where the size of the binary polynomials equals the word-size w of the LEON-2 processor, namely 32 bit.

Two principal implementations of ECC operations have been employed for evaluation of the merits of the proposed multiply step instructions. One used the left-to-right comb method with a look-up table containing 16 entries, as mentioned in Section 3, for polynomial multiplication and shift and XOR instructions for reduction. This implementation was tailored to the use of GF(2^{191}) with the above reduction polynomial and therefore especially suited for constrained client devices. The different variants used for evaluation are denoted with the prefix

OPT in the rest of this text. The second implementation could work in a binary extension field of arbitrary length with any reduction polynomial, which fulfills the following requirement: It may only have non-zero coefficients for powers $< w$. Such an implementation is favorable for server machines in mobile and wireless environments. The variants are based on the MULGF2 operation as a building block for all GF(2^m) multiplication, squaring and reduction. They vary only in the implementation of the MULGF2 operation and are denoted with the prefix FLEX. All OPT and FLEX implementations used the method described by López and Dahab to perform an elliptic scalar multiplication [10].

6.1 Running Times

Table 1 presents the running times of multiplication and squaring in GF(2^{191}) and of a complete elliptic scalar multiplication for the three variants of the flexible implementation. The running time is measured in clock cycles. The first column (FLEX1) gives the results for the pure software variant, where the MULGF2 operation has been implemented with shift and XOR instructions. The second and third column list the running times for adapted versions, where the word-size polynomial multiplication (MULGF2 operation) has been optimized. FLEX2 refers to the variant which made use of the `MULGFS` instruction as described in Section 5.1. The results for FLEX3 are for an implementation which utilizes both `MULGFS` and `MULGFS2` instructions as outlined in Section 5.2. Both FLEX2 and FLEX3 necessitated only minor changes to the code of FLEX1.

Table 1. Execution times of important operations for ECC over GF(2^{191}) for the FLEX variants in clock cycles

	FLEX1	FLEX2	FLEX3
	Software	`MULGFS` instr.	`MULGFS` and `MULGFS2` instr.
GF(2^{191}) multiplication	15,344	2,306	1,620
GF(2^{191}) squaring	5,335	691	476
EC scalar multiplication	22,485,650	3,260,478	2,319,558

The running times for the EC scalar multiplication from Table 1 are a representative measure to compare the overall speed of the three implementations. The use of the `MULGFS` instruction alone (FLEX2) yields a speedup factor of nearly 7 over the pure software version. If both multiply step instructions are available (FLEX3), the speedup factor is nearly 10. Note that squaring is a linear operation and therefore performs much faster than multiplication.

The optimized implementation in pure software (OPT1) can be enhanced with the proposed multiply step instructions. GF(2^{191}) multiplication which uses the `MULGFS` and `MULGFS2` instructions is faster than the multiplication of the original software implementation. Table 2 lists the running times of the three versions, where OPT2 uses just the `MULGFS` instruction and OPT3 makes use of both `MULGFS` and `MULGFS2` instructions to speed up GF(2^{191}) multiplication.

Table 2. Execution times of important operations for ECC over GF(2^{191}) for the OPT variants in clock cycles

	OPT1	OPT2	OPT3
	Software	`MULGFS` instr.	`MULGFS` and `MULGFS2` instr.
GF(2^{191}) multiplication	3,182	2,076	1,500
GF(2^{191}) squaring	273	273	273
EC scalar multiplication	3,909,690	2,706,560	2,054,282

Table 3. Memory requirement of the OPT and FLEX variants of elliptic scalar multiplication in bytes

	OPT1	OPT3	FLEX1	FLEX3
Code section size	4,928	2,920	3,904	2,592
Data section size	1,024	1.024	264	264
Total size	5,952	3,944	4,168	2,856
Additional RAM usage	384	none	none	none

Note that the running time for the GF(2^{191}) multiplication for OPT2 and OPT3 are smaller than those of FLEX2 and FLEX3 because the former use a reduction step which is tailored to the reduction polynomial $t^{191} + t^9 + 1$. EC scalar multiplication is sped up by about 45% with the `MULGFS` instruction and by 90% through the use of both `MULGFS` and `MULGFS2` instructions. Additionally, FLEX2 is about 15% faster than OPT1 and FLEX3 is about 40% faster.

6.2 Memory Requirements

Table 3 compares the size of the code and data sections of an SPARC executable which implements the full EC scalar multiplication for the OPT and FLEX variants. The executables have been obtained by linking the object files for each implementation without linking standard library routines. The size of the code and data sections have subsequently been dumped with the GNU *objdump* tool. As the values for OPT2 and OPT3 and those for FLEX2 and FLEX3 are nearly identical, only one implementation of each group has been listed exemplarily.

The executables of the FLEX2 and FLEX3 implementations are only half the size of OPT1. This is mainly because OPT1 uses a hard-coded look-up table for squaring and also features larger subroutines. OPT2 and OPT3 have 70% smaller code sections and a 50% smaller executable compared to OPT1. In addition, OPT1 uses an look-up table for GF(2^{191}) multiplication which is calculated on-the-fly and requires additional space in the RAM. This memory requirement is eliminated in OPT2, OPT3 and all FLEX variants.

The costs of additional hardware for implementation of both multiply step instructions have been evaluated by comparing the synthesis results for the different processor versions. The enhanced version had an increase in size of less than 1% and is therefore negligible.

The OPT variants are the most likely candidates for usage in devices which are constrained regarding their energy supply. To compare OPT1 with the enhanced versions OPT2 and OPT3, it is important to note that load and store instructions normally require more energy than other instructions on a common microprocessor; see e.g. the work of Sinha et al. [15]. Based on that fact it can be established that OPT2 and OPT3 have a better energy efficiency than OPT1 for two reasons: They have shorter running times and do not use as many load and store instructions, as they perform no table look-ups for field multiplication.

7 Conclusions

In this paper we presented an extension to general-purpose processors which speeds up ECC over GF(2^m). The use of multiply step instructions accelerates multiplication of binary polynomials, i.e. the MULGF2 operation, which can be used to realize arithmetic operations in GF(2^m) in an efficient manner. We have integrated both proposed versions of the multiply step instruction into a SPARC V8-compliant processor core. Two different ECC implementations have been accelerated through the use of our instructions. The implementation optimized for GF(2^{191}) and a fixed reduction polynomial has been sped up by 90% while reducing the size of its executable and its RAM usage. The flexible implementation, which could cater for different fields lengths m and an important set of reduction polynomials, was accelerated by an factor of over 10. Additionally, the enhanced flexible version could outperform the original optimized implementation by 40%. All enhancements required only minor changes to the software code of the ECC implementations.

We have discussed the merits of our enhancements for both constrained devices and server machines in a security-enhanced mobile and wireless environment. The benefits for devices constrained in available die size and memory seem especially significant, as our multiply step instructions require little additional hardware and reduce memory demand regarding both code size and runtime RAM requirements. Additionally, the implementations which use our instructions are likely to be more energy efficient on common general-purpose processors.

Acknowledgements. The research described in this paper was supported by the Austrian Science Fund (FWF) under grant number P16952-N04 (“Instruction Set Extensions for Public-Key Cryptography”).

References

1. American National Standards Institute (ANSI). X9.62-1998, Public key cryptography for the financial services industry: The elliptic curve digital signature algorithm (ECDSA), Jan. 1999.
2. I. F. Blake, G. Seroussi, and N. P. Smart. *Elliptic Curves in Cryptography.* Cambridge University Press, 1999.

3. A. Chandrakasan, W. Bowhill, and F. Fox. *Design of High-Performance Microprocessor Circuits*. IEEE Press, 2001.
4. J. Gaisler. The LEON-2 Processor User's Manual (Version 1.0.10). Available for download at `http://www.gaisler.com/doc/leon2-1.0.10.pdf`, Jan. 2003.
5. J. Großschädl and G.-A. Kamendje. Instruction set extension for fast elliptic curve cryptography over binary finite fields GF(2^m). In *Proceedings of the 14th IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP 2003)*, pp. 455–468. IEEE Computer Society Press, 2003.
6. J. Großschädl and G.-A. Kamendje. Low-power design of a functional unit for arithmetic in finite fields GF(p) and GF(2^m). In *Information Security Applications*, vol. 2908 of *Lecture Notes in Computer Science*, pp. 227–243. Springer Verlag, 2003.
7. D. Hankerson, J. López Hernandez, and A. J. Menezes. Software implementation of elliptic curve cryptography over binary fields. In *Cryptographic Hardware and Embedded Systems — CHES 2000*, vol. 1965 of *Lecture Notes in Computer Science*, pp. 1–24. Springer Verlag, 2000.
8. D. Hankerson, A. Menezes, and S. Vanstone. *Guide to Elliptic Curve Cryptography*. Springer Verlag, 2004.
9. Ç. K. Koç and T. Acar. Montgomery multiplication in GF(2^k). *Designs, Codes and Cryptography*, 14(1):57–69, Apr. 1998.
10. J. López and R. Dahab. Fast multiplication on elliptic curves over $GF(2^m)$ without precomputation. In *Cryptographic Hardware and Embedded Systems*, vol. 1717 of *Lecture Notes in Computer Science*, pp. 316–327. Springer Verlag, 1999.
11. J. López and R. Dahab. High-speed software multiplication in $\mathbb{F}_{2^m}$. In *Progress in Cryptology — INDOCRYPT 2000*, vol. 1977 of *Lecture Notes in Computer Science*, pp. 203–212. Springer Verlag, 2000.
12. E. Nahum, S. O'Malley, H. Orman, and R. Schroeppel. Towards high performance cryptographic software. In *Proceedings of the 3rd IEEE Workshop on the Architecture and Implementation of High Performance Communication Subsystems (HPCS '95)*, pp. 69–72. IEEE, 1995.
13. National Institute of Standards and Technology (NIST). Digital Signature Standard (DSS). Federal Information Processing Standards Publication 186-2, 2000.
14. R. Schroeppel, H. Orman, S. O'Malley, and O. Spatscheck. Fast key exchange with elliptic curve systems. In *Advances in Cryptology — CRYPTO '95*, vol. 963 of *Lecture Notes in Computer Science*, pp. 43–56. Springer Verlag, 1995.
15. A. Sinha and A. Chandrakasan. Jouletrack – A web based tool for software energy profiling. In *Proceedings of the 38th Design Automation Conference (DAC 2001)*, pp. 220–225. ACM Press, 2001.
16. SPARC International, Inc. The SPARC Architecture Manual Version 8. Available for download at `http://www.sparc.org/standards/V8.pdf`, Aug. 1993.
17. A. Weimerskirch, D. Stebila, and S. Chang Shantz. Generic GF(2^m) arithmetic in software and its application to ECC. In *Information Security and Privacy — ACISP 2003*, vol. 2727 of *Lecture Notes in Computer Science*, pp. 79–92. Springer Verlag, 2003.
18. XESS Corporation. XSV-800 Virtex Prototyping Board with 2.5V, 800,000-gate FPGA. Product brief, available online at `http://www.xess.com/prod014_4.php3`, 2001.

Scalable Design Framework for JPEG2000 System Architecture

Hiroshi Tsutsui[1], Takahiko Masuzaki[1], Yoshiteru Hayashi[1], Yoshitaka Taki[1], Tomonori Izumi[1], Takao Onoye[2], and Yukihiro Nakamura[1]

[1] Department of Communications and Computer Engineering, Kyoto University
Yoshida-hon-machi, Sakyo, Kyoto, 606-8501 Japan
{tsutsui, masuz, teru, taki}@easter.kuee.kyoto-u.ac.jp,
{izumi, nakamura}@kuee.kyoto-u.ac.jp

[2] Department of Information Systems Engineering, Osaka University
2-1 Yamada-Oka, Suita, Osaka, 565-0871 Japan
onoye@ist.osaka-u.ac.jp

Abstract. For the exploration of system architecture dedicated to JPEG2000 coding, decoding and codec, a novel design framework is constructed. In order to utilize the scalability of JPEG2000 algorithm aggressively in system implementation, three types of modules are prepared for JPEG2000 coding/decoding/codec procedures, i.e. software, software accelerated with user-defined instructions, and dedicated hardware. Specifically, dedicated hardware modules for forward and inverse discrete wavelet transformation (shortly DWT), entropy coder, entropy decoder, and entropy codec as well as software acceleration of DWT procedure are devised to be used in the framework. Furthermore, a JPEG2000 encoder LSI, which consists of a configurable processor Xtensa, the DWT module, and the entropy coder, is fabricated to exemplify the system implementation designed through the use of proposed framework.

1 Introduction

The increasing use of multimedia information requires image coding system to compress different types of still images with different characteristics by a single processing flow besides attaining high coding efficiency. To fulfill this requirement, JPEG2000 is currently being developed by ISO/ IEC JTC1/SC29 WG1 (commonly known as the JPEG), and JPEG2000 Part I[1] was standardized in January, 2001. Distinctively, in JPEG2000, discrete wavelet transformation (shortly DWT) is adopted to decorrelate images spatially to improve compression efficiency. With the use of this transformation, so-called *embedded stream* can be generated, in which code for low quality/bitrate image is included in that for high quality/bitrate image. Therefore, JPEG2000 can be regarded as the viable image coding scheme in the coming network era to be used in a variety of terminals for different application fields. However, this fact also implies that performance requirement for terminals and applications varies widely. Thus any of single software or hardware implementation can hardly fulfill performance

P.-C. Yew and J. Xue (Eds.): ACSAC 2004, LNCS 3189, pp. 296–308, 2004.

requirements for all range of terminals and applications. On the other hand, it is also impossible to arrange a full set of customized implementations for all of terminals or applications in terms of man-power resources.

Motivated by this tendency, we propose a novel framework of JPEG2000 system architecture, providing the distinctive ability of architectural exploration in accordance with the specification of each terminal. Through the use of this framework, an efficient JPEG2000 system organization is obtained by referring to performance requirements and limitations for implementation. The proposed framework is based on Tensilica's configurable processor Xtensa[2], which has an ability to be customized for a specific application by equipping user-defined instructions described in Tensilica Instruction Extension (TIE) language[3]. Enhancing this distinctive feature to such an extent to equip specific hardware modules prepared for procedures in JPEG2000, our framework provides scalable solution for JPEG2000 system architecture.

For each procedure of JPEG2000 coding and/or decoding, either of software implementation, software implementation accelerated by user-defined instructions, or hardware implementation is selectively employed. The implementation scheme for all procedures are decided by referring to performance requirements and/or limitations to design. In case extremely high processing performance is needed far more than that of hardware implementation, it is possible to equip two or more modules at the same time.

2 JPEG2000 Processing Flow

Fig. 1 depicts the procedures of JPEG2000 encoding scheme. First, a target image is divided into square regions, called *tiles*. Tiles of each color component are called *tile components*. Then 2-D forward DWT decomposes a tile component into LL, HL, LH, and HH subbands by applying 1-D forward DWT to a tile component vertically and horizontally. The low resolution version of the original tile component, i.e. LL subband, is to be decomposed into four subbands recursively. A subband is divided into *code-blocks*, each of which is coded individually by entropy coder. The entropy coder adopted in JPEG2000 is context-based adaptive binary arithmetic coder which consists of coefficient bit modeling process to generate contexts and arithmetic coding process, known as MQ-coder, to compress a binary sequence based on the context of each bit.

Decoding scheme of JPEG2000 is the reverse process of the encoding. During this process, 2-D backward DWT is realized by applying a series of 1-D backward DWTs (horizontal 1-D DWT and vertical 1-D DWT) to a tile component also in the reverse order of 2-D forward DWT. The set of filter coefficients used in 1-D backward DWT is the same as that in the forward DWT. Fig. 2 depicts the entropy coding and decoding procedure. Coefficient bit modeling is a common process to encoding and decoding. MQ-decoder extracts binary sequences from compressed data referring to contexts generated by the coefficient bit modeling process, while MQ-coder generates compressed data from contexts and binary sequences.

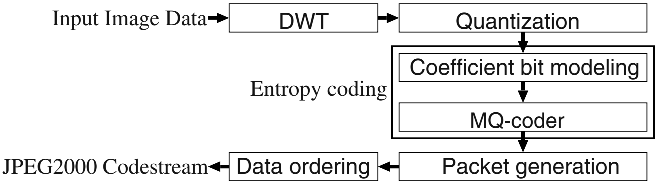
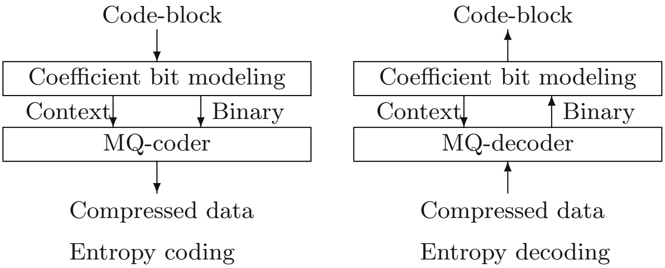


Fig. 1. Block diagram of JPEG2000 encoding



Entropy coding

Entropy decoding

Fig. 2. Entropy coding and decoding

3 JPEG2000 System Framework

Our proposed framework is distinctive in that an efficient JPEG2000 encoding and decoding system architecture can be explored by selecting implementation scheme for each procedure considering performance requirements such as image resolution or processing throughput and limitations to design such as power consumption, process technology rule, or chip size. Three types of implementation schemes are prepared in our framework; software implementation, software implementation accelerated by user-defined instructions added to Xtensa, and dedicated hardware implementation. As for hardware implementation, multiple modules can be used at the same time if necessary. To embody such a Plug-and-Play like feature, each hardware module is designed to have a generic SRAM-based interface which can support various bus architectures by only designing interface converters. Therefore, our framework makes it much easier to design a JPEG2000 encoding and decoding system than conventional tedious manual design tasks of each procedure, which would be implemented as software or hardware. For forward/backward DWT and entropy coding/decoding, common hardware components are prepared. These procedures handle relatively large square regions of an image, called tile and code-block, respectively. Such a procedure is inherently suitable as a component, and the overhead of data transfer between the hardware component and memory can be concealed when direct memory access (DMA) is applied effectively. In addition to these common hardware components, an software module accelerated by user-defined instructions is prepared for DWT. Since the dominant operation in DWT procedure is filter operation, a set of instructions for filter operation is added to the Xtensa's

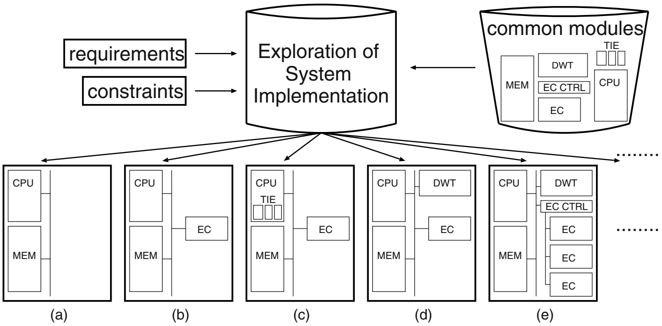


Fig. 3. Basic structure of the proposed framework

original instruction set so that considerable performance improvement can be achieved without any additional hardware component.

Moreover, to organize JPEG2000 codecs, an entropy codec hardware component is prepared. This component is smaller than the simple combination of the entropy coder and decoder in terms of the number of gates. As for DWT, the differences between encoding and decoding are ordering of two 1-D DWTs applied in series, ordering of filter coefficients, and signs of several filter coefficients. The first one seems to affect largely the DWT architecture. However, in the case of lossy compression, the error introduced by employing same DWT ordering for both forward and backward DWTs is quite slight. Therefore we employ the same ordering of DWTs to make it easy to design forward and/or backward 2-D DWT. As for other differences, we simply introduce multiplexers to select the signs of filter coefficients, etc.

Fig. 3 illustrates the basic structure of the proposed framework. Each system implementation exemplified in Fig. 3 is briefed below. Though only the case for JPEG2000 encoders is explained, JPEG2000 decoders and codecs can be implemented by the same way of the encoders.

Fig. 3(a) is a system implementation example where all modules are implemented as software so that the system is composed only of a CPU and a main memory. This solution is used when it is impossible to employ hardware modules to the system due to die size limitation, the processing speed is not the key point aimed at, or the CPU provides sufficient performance for application.

Fig. 3(b) is a system implementation example where only the entropy coding (shortly EC in the figure) procedure is implemented by a hardware module. The hardware implementation of the entropy coding makes a large contribution to improvement of the speed, since this procedure is the most computationally intensive one among all the procedures as its detailed discussion is given in the next section.

Fig. 3(c) is a system implementation example where the entropy coder is implemented by hardware module, and instructions for DWT's filter operations are added to Xtensa. Since DWT procedure handles a whole tile, the number of cycles needed to execute the procedure in DWT, except for filter operations, such as address calculation, memory accessing, and so forth, is still large. Performance improvement in comparison with Fig. 3(b), however, can be achieved without any additional hardware module.

Fig. 3(d) is a system implementation example where both the entropy coder and DWT are implemented as hardware modules. In addition to the performance improvement as the benefit of these modules, memory accessing bandwidth is reduced.

Fig. 3(e) is a system implementation example where DWT is implemented by hardware module, and the entropy coding procedure is implemented by multiple hardware modules. This solution is the fastest among Fig. 3(a) through (e), and attains very high throughput. In this case, an additional controller is needed to manage these modules.

In this manner, our framework successfully provides scalable solution for JPEG2000 system architecture with the use of common modules.

4 Analysis of JPEG2000 Encoding

To construct the framework, first we implemented software JPEG2000 encoder and decoder, and analyzed computational costs of all procedures. Since DWT is to be executed in fixed point arithmetic in our software, even embedded CPUs which has no floating-point unit (FPU) can execute this software without any additional cycles needed for floating arithmetic emulation. The main specifications of our software are summarized in Table 1, and the result of profiling encoding process by the instruction set simulator (shortly ISS) of the target CPU Xtensa using a test image LENA is summarized in Table 2. The function `encode` is to encode whole image and does not include the routines for input/output from/to a file. In the function `entropy_coding`, the entropy coding including coefficient bit modeling and arithmetic coding by MQ-coder is executed. The function `FDWT_97`

Table 1. Main specification of the JPEG2000 software

tile size	128 × 128
DWT	9/7 irreversible filter
DWT decomposition level	3
code-block size	32 × 32

Table 2. Result of profiling encoding process

%	function name	self cycle	child cycle	total cycle	call
99.5	`main`	0.13	1371191.89	1371192.02	1
96.3	`encode`	540.26	1326074.65	1326614.91	1
64.4	`entropy_coding`	84.50	887680.17	887764.67	304
20.2	`FDWT_97`	29298.20	249056.71	278354.91	16
14.0	`FILTD_97`	191668.46	771.41	192439.87	7168

unit of number of cycles is Kcycle

is to execute DWT on a tile component and includes FILTD_97 for 1-D DWT on an array.

According to this table, it can be said that the coefficient bit modeling and arithmetic coding procedures occupy 64.4% of total encoding cycles, DWT procedure occupies 20.2% of total encoding cycles, and FILTD_97 function occupies 69.1% of DWT processing cycles.

5 JPEG2000 Processing Modules

5.1 DWT Module

DWT hardware. When implementing DWT as dedicated hardware, the essential factor to be considered is memory organization for storing intermediate data during recursive filter processing. There are two methods to store intermediate data for DWT. One is to store the data to the main memory or a tile buffer which is placed in a DWT hardware module. In this case, whenever vertical and horizontal DWT is attempted, the data is read from the memory/buffer and the transformed coefficients are written back to it. The other is the so-called line-based method[4] which is to store the data to a line buffer containing several lines in a tile. In this case, vertical and horizontal DWT can be done at the same time by utilizing the line buffer, so that this method requests less amount of data transfer over a bus than that using the main memory. Thus, we adopted line-based method to implement DWT hardware module.

To implement DWT filters, we adopted straightforward finite impulse response (FIR) filter, where, for calculating each transformed coefficient, weighted addition of a coefficient sequence with the filter length is executed. Assuming that the depth of input image is 8-bit, this module can implement one level 2-D DWT over a tile whose width is 128 or less. The tile size of 128×128 is reasonable since it is adopted in JPEG2000 Profile 0[5], which is intended mainly for hardware implementation.

The architecture of our DWT hardware module is shown in Fig. 4, which consists of a core module, a line buffer, and an IO controller. This core module comprises the following sub-modules; an extension module which extends a sequence of coefficients at the edges by the so-called periodic symmetric extension procedure, low-pass FIR filters whose filter length is 9, high-pass FIR filters whose filter length is 7, and a pair of shift registers which receive output data from vertical DWT and store vertically low-passed and high-passed 9 coefficients to be fed to horizontal DWT.

The line buffer consists of 13 line memories each of which is a 15-bit 128-word memory. The number of the additional 7 bits for the input bit depth, 8 bits, is large enough to realize as high precision as floating point operations. Totally 9 coefficients, all of which belong to the identical column of an image, are loaded to the core module from 9 lines of the line buffer every clock. Other 2 lines of the line buffer are used for receiving transformed coefficients from the core modules every clock. The other 2 lines of the line buffer are used for IO access between the DWT and CPU.

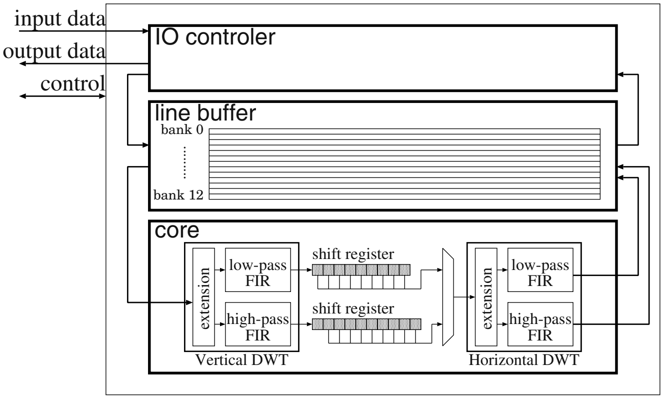


Fig. 4. Architecture of our DWT module

This core module works as follows. The 9 coefficients from the line buffer are transformed by the vertical DWT module and the results, i.e. low-passed and high-passed coefficients, are stored into the shift registers. At the same time, horizontal DWTs of vertically high-passed and low-passed sequences stored in shift registers are executed alternately, so that LL, HL, HH, and HH coefficients are output from the core module every 2 cycles.

As mentioned before, we employ the same 1-D DWT ordering for both forward and backward 2-D DWT, so as to make the architecture of forward and backward DWT almost identical. The proposed architecture of forward DWT module is designed through the use of Verilog-HDL. When this hardware module is used, the number of cycles needed for DWT with a test image LENA is 0.215 Mcycles, which is only 0.077 % of 278 Mcycles needed for software DWT. The DWT module is synthesized into 17,650 gates by using Synopsys's Design Compiler with 0.18 μm CMOS technology, with its critical path delay of 12 nsec.

Accelerated DWT software. According to the result of profiling, multiplications of fixed point variables and filter coefficients of Table 3; α, β, γ, δ, K, and $1/K$; need 6961.86 Kcycles, 7787.56 Kcycles, 7502.93 Kcycles, 6930.11 Kcycles, 6775.10 Kcycles, and 6971.17 Kcycles, respectively through entire image. Total of these values occupy 22.3% of execution cycles needed for function FILTD_97. Thus we implement these multiplications by user-defined instructions described in TIE. The circuits of the instructions consist of shifters and adders.

Custom instructions MUL_A, MUL_B, MUL_C, MUL_D, MUL_K, and MUL_R are to multiply positive input by lifting constants; α, β, γ, δ, K, and $1/K$, respectively. SMUL_A, SMUL_B, SMUL_C and SMUL_D are extended version of MUL_A, MUL_B, MUL_C and MUL_D, which can multiply negative input as well as positive input by lifting constants. At the final stage of lifting, target values must be shifted in right and rounded. These operations are also implemented as custom instructions. SMUL_K and SMUL_R are extended version of MUL_K and MUL_R. Same as SMUL_[A-D], these

Table 3. Filter coefficiets

α	-1.586 134 342 059 924
β	-0.052 980 118 572 961
γ	0.882 911 075 530 934
δ	0.443 506 852 043 971
K	1.230 174 104 914 001

Table 4. The number of gates of user-defined instructions

instruction	#gate
MUL_A	551.75
MUL_B	401.00
MUL_C	510.25
MUL_D	489.50
MUL_K	570.50
MUL_R	367.00
other	254.25
total	3144.25

Table 5. The number of gates of user-defined instructions

instruction	#gate
SMUL_A	1085.25
SMUL_B	739.25
SMUL_C	947.00
SMUL_D	1009.25
SMUL_K	1698.25
SMUL_R	1039.00
other	439.75
total	6957.75

instructions can handle negative values with equipping a right shifter and a rounding circuit.

The result of simulation by ISS using a test image LENA shows that when MUL_A, MUL_B, MUL_C, MUL_D, MUL_K, and MUL_R are equipped, the number of cycles to call FILTD_97 function is reduced to 175.564 Mcycles (88.6%) from 192.440 Mcycles, and that when SMUL_A, SMUL_B, SMUL_C, SMUL_D, SMUL_K, and SMUL_R are implemented, the number of cycles to call FILTD_97 function can be reduced to 133.461 Mcycles (69.4%) from 192.440 Mcycles.

According to the synthesis results attained by the same manner as the hardware DWT module, the critical path delay of MUL_A, MUL_B, MUL_C, MUL_D, MUL_K and MUL_R is 6.2 nsec, ant that of SMUL_A, SMUL_B, SMUL_C, SMUL_D, SMUL_K and SMUL_R is 9.5 nsec, The numbers of gates of custom instructions are summarized in Table 4 and 5.

5.2 Entropy Coder, Decoder, and Codec

There are two reasons why the entropy coding of JPEG2000 incurs such a high computational cost. One is that a context of a coefficient on a bit plane depends on sign bits, significant states, and some reference states of the eight nearest-neighbor coefficients. Therefore, there are many conditional branches and many operations which crop variables into a bit. The other is that MQ-coder updates its internal state after compression of every one binary symbol. When JPEG2000 entropy coder is implemented as hardware, MQ-coder may become the performance bottle-neck of the total system since the MQ-coder requires at least 1 cycle to process 1 binary symbol. Needless to say, considering hardware utilization efficiency coefficient bit modeling must be implemented with the same throughput as the MQ-coder. In our hardware entropy coder, pixel skipping scheme[6] is used to attain almost ideal performance.

Entropy Coder. Fig. 5 depicts the block diagram of our entropy coder, which consists of a coefficient bit modeling module, an MQ-coder, an IO controller, a plane controller, an FIFO, and a set of buffer memories. As buffer memories, the entropy coder has a code-block buffer to store code-block data, plane buffers to store bit plane as well as reference data needed to generate contexts, and stream buffer to store encoded data called *stream*. The FIFO is used to suppress the difference of the throughputs of the modeling module and the MQ-coder.

The above mentioned entropy coder is designed by Verilog-HDL. The result of logic simulation using sample image LENA indicates that when this hardware entropy coder is employed, the number of cycles needed for entropy coding of whole image is about 3.20 Mcycles, i.e. only 0.360 % of 887.76 Mcycles needed for software entropy coding.

The critical path delay is 7.0 nsec which is concluded by synthesis the entropy coder with 0.18 μm CMOS technology. The gate counts for this module is 7,901.

Next, let us discuss the size of memory for the entropy coder. The coder requests 12 bit $\times$ (32×32) = 12,288 bit as the code-block buffer, $4 \times (32 \times 32) = 4{,}096$ bit as 4 plane buffers, and $2 \times (32 \times 32) = 2{,}048$ bit as the double-buffered bit plane buffer. Here, the code-block buffer must have 12 bit depth in the case that the bit depth of input image is 8, and the number of guard bits is 2, which is enough to avoid the overflow of the result of DWT. As for the stream buffer for output, 8,192 bit is large enough when quantizer's step size equals 1, according to the software simulation. Consequently, 26,624 bit of memory elements are needed in total. It must be noticed that the size of stream buffer can be reduced when implementing it as an FIFO.

Entropy Decoder. Fig. 6 depicts the block diagram of our entropy decoder, which is similarly organized as the entropy coder. The differences are that the decoder does not equip FIFO, whereas controllers, MQ-decoder, and coefficient bit modeling module of the decoder are customized for decoding. Our MQ-decoder returns a binary to coefficient bit modeling module in 1cycle.

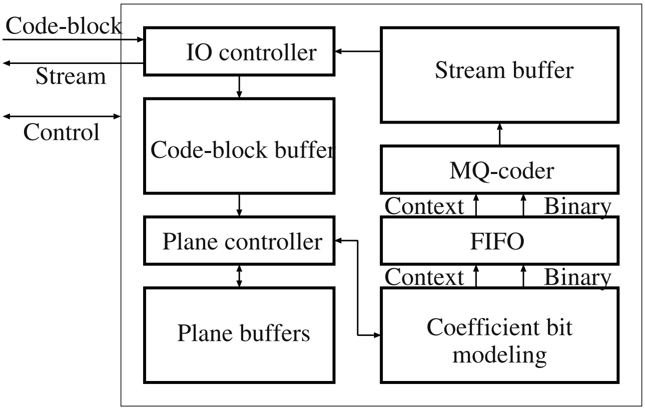


Fig. 5. Architecture of our entropy coder

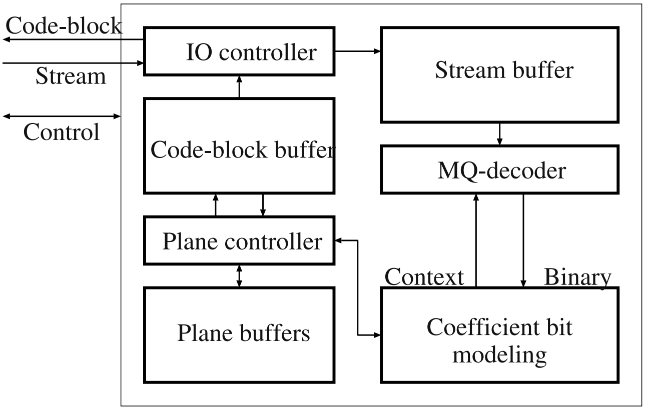


Fig. 6. Architecture of our entropy decoder

The entropy decoder is designed through the use of Verilog-HDL. The result of logic simulation using sample image LENA indicates that when this hardware entropy decoder is employed, the number of cycles needed for entropy coding of whole image is about 5.03 Mcycles, i.e. only 0.283 % of 1776 Mcycles needed for software entropy decoding.

The critical path delay is 7.0 nsec which is concluded by synthesis the entropy decoder with 0.18 μm CMOS technology. The gate counts for this module is 7,901.

Entropy Codec. In our framework, an entropy codec is also prepared in addition of the above mentioned entropy coder and decoder. By sharing some part of circuits between MQ-coder and MQ-decoder and the circuits to generate contexts of coefficient bit modeling for coding and decoding, we can successfully reduce

Table 6. Comparison of the number of gates between entropy coder, decoder and codec

	Submodule	#gate	total
Entropy coder	MQ-coder	2,983	5,658
	Coefficient bit modeling	2,675	
Entropy decoder	MQ-decoder	3,881	6,454
	Coefficient bit modeling	2,645	
Entropy codec	MQ-codec	6,048	9,713
	Coefficient bit modeling	3,665	

Table 7. Main features of the LSI

Technology	Hitachi 0.18μm CMOS
Interconnect	5 metal layers, PolySi
Power supply	1.8V
Chip area	5.9 × 5.9 mm^2
Design method	Standard-cell-based
Package	256pin BGA

Table 8. The numbers of gates and memory bits

#gate of Xtensa	44,000
#gate of DWT	17,650
#gate of Entropy coder	10,207
#bit of FF in DWT module	24,960
#bit of FF in Entropy coder	6,144
#bit of RAM of Entropy coder	65,536

#gate dose not include the gates of FF used as memory.

Table 9. The number of cycles

function name	total cycle by software	total cycle by using this LSI
encode	1326.6 (100%)	163.8 (12%)
entropy_coding	887.8 (67%)	3.20 (0.36%)
FDWT_97	278.4 (21%)	0.215 (0.08%)

Unit of number of cycles is Mcycle.
% of total cycle by software means the ratio to cycle of encode.
% of total cycle by using this LSI means the ratio to cycle by software.

the number of gates of the entropy codec with maintaining its performance. The comparison of the numbers of gates required for the dominant parts, which are MQ-coder/decoder and coefficient bit modeling, among entropy coder, decoder and codec is summarized in Table 6. The number of gates for MQ-codec is 88% of that for the combination of MQ-coder and MQ-decoder, and the number of gates of coefficient bit modeling for codec is 69% of that for the combination of those for encoding and decoding.

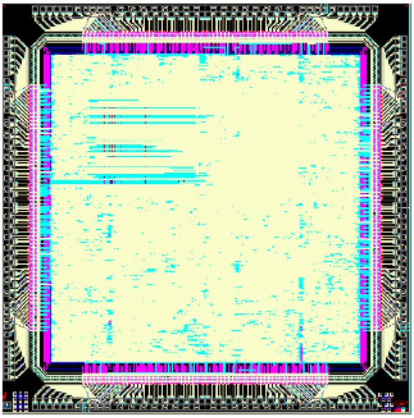


Fig. 7. Layout patterns for the LSI

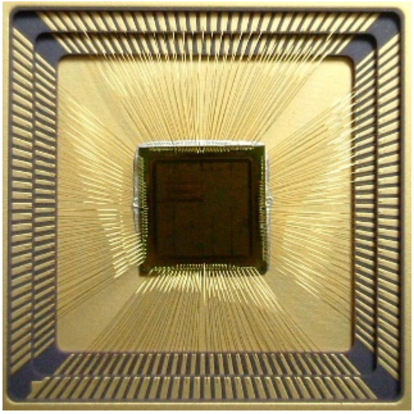


Fig. 8. Photograph of the LSI

6 LSI Implementation Result

In order to demonstrate the practicability of the proposed framework, we fabricated an JPEG2000 encoder LSI, which consists of our DWT hardware module, hardware entropy coder module, and Xtensa. A photograph of the LSI and layout patterns attained for the LSI are shown in Figs. 8 and 7, respectively.

Table 7 summarizes the specifications of the fabricated LSI. The LSI is with 1-Kword × 32-bit single port RAM as the code-block buffer and stream buffer of the entropy coder. The line buffer of DWT module and the plane buffers of entropy coding module are implemented by flip-flop (FF) arrays. The numbers of gates and memory bits are summarized in Table 8. The critical path delay of this LSI is 18 nsec, which assures 55.5 MHz operation.

The comparison between the number of cycles needed to encode the test image LENA by software and that by using this LSI is summarized in Table 9.

7 Conclusion

In this paper, a novel design framework to realize an efficient implementation of JPEG2000 encoder, decoder, and codec in accordance with the requirements and constraints of each terminals and applications has been proposed. This framework is distinctive in that for each procedure of JPEG2000 coding system, implementation scheme can be selected among software implementation, software implementation accelerated with user-defined instructions, and dedicated hardware implementation, so as to optimize the system organization. To demonstrate the practicability of the framework, we fabricated an LSI to exemplify a generated system implementation, in which our DWT hardware module and hardware entropy coder module were implemented with configurable processor Xtensa.

Acknowledgement. The VLSI chip in this study has been fabricated in the chip fabrication program of VLSI Design and Education Center (VDEC), the University of Tokyo with the collaboration by Hitachi Ltd. and Dai Nippon Printing Corporation.

References

1. ISO/IEC JTC1/SC29/WG1, "Information technology – JPEG2000 image coding system: Core coding system," Oct. 2002.
2. Tensilica, Inc., *Xtensa Application Specific Microprocessor Solutions — Overview Handbook*, Sept. 2000.
3. Tensilica, Inc., *Tensilica Instruction Extension (TIE) Language — User's Guide*, Sept. 2000.
4. ISO/IEC JTC1/SC29/WG1, "JPEG2000 verification model 9.1 (technical description)," June 2001.
5. ISO/IEC JTC1/SC29/WG1, "Draft of FPDRAM-1 to 15444-1," Dec. 2000.
6. Kuan-Fu Chen, Chung-Jr Lian, Hong-Hui Chen, and Liang-Gee Chen, "Analysis and architecture design of EBCOT for JPEG-2000," in *Proc. of the 2001 IEEE International Symposium on Circuits and Systems (ISCAS 2001)*, Vol. 2, pp. 765–768, Mar. 2001.

Real-Time Three Dimensional Vision

JongSu Yi[1], JunSeong Kim[1], LiPing Li[2], John Morris[1,3], Gareth Lee[4] and Philippe Leclercq[5]

[1] School of Electrical and Electronics Engineering,
Chung-Ang University, Seoul 156-756, Korea

[2] Department of Computer Science, Harbin Normal University, Harbin, China 150080

[3] Department of Electrical and Electronic Engineering,
The University of Auckland, New Zealand

[4] Department of Computer Science and Software Engineering,
University of Western Australia, Nedlands, WA 6009, Australia

[5] Department of Electrical and Computer Engineering,
University of Western Australia, Nedlands, WA 6009, Australia

Abstract. Active systems for collision avoidance in 'noisy' environments such as traffic which contain large numbers of moving objects will be subject to considerable interference when the majority of the moving objects are equipped with common avoidance systems. Thus passive systems, which require only input from the environment, are the best candidates for this task. In this paper, we investigate the feasibility of real-time stereo vision for collision avoidance systems. Software simulations have determined that sum-of-absolute-difference correlation techniques match well but hardware accelerators are necessary to generate depth maps at camera frame rates. Regular structures, linear data flow and abundant parallelism make correlation algorithms good candidates for reconfigurable hardware. The SAD cost function requires only adders and comparators for which modern FPGAs provide good support. However accurate depth maps require large disparity ranges and high resolution images and fitting a full correlator on a single FPGA becomes a challenge. We implemented SAD algorithms in VHDL and synthesized them to determine resource requirements and performance. Altering the shape of the correlation window to reduce its height compared to its width reduces storage requirements with negligible effects on matching accuracy. Models which used the internal block memory provided by modern FPGAs to store the 'inactive' portions of scan lines were compared with simpler models which used the logic cell flip-flops. From these results, we have developed a simple predictor which enables one to rapidly determine whether a target appliction is feasible.

1 Introduction

A collision avoidance system for any mobile device - from a robot to a large vehicle - requires the ability to build a three-dimensional 'map' of its environment. Traditionally this has been accomplished by active sensors which send a pulse - either electromagnetic or sonar - and detect the reflected return. Such active

P.-C. Yew and J. Xue (Eds.): ACSAC 2004, LNCS 3189, pp. 309–320, 2004.

systems work well in low density environments where the number of moving vehicles is small and the probability that active sensors will interfere is small, so that simple techniques prevent a sensor from being confused by sensing pulses from other vehicles. For example, radar systems which detect impending aircraft collisions appear to be effective as do ultrasonic systems in small groups of robots. However, when the density of autonomous - and potentially colliding - objects becomes large, active systems create ‘noisy’ environments. Heavy traffic brings large numbers of vehicles with a wide range of speeds and directions into close proximity. If all vehicles were equipped with active sensors, distinguishing between extremely weak reflections and primary pulses from distant vehicles will present a daunting - and potentially insurmountable - problem for intelligent sensors. Passive systems, on the other hand, are much less sensitive to environmental interference. Stereo vision - or the use of pairs of images taken from different viewpoints - is able to provide detailed three-dimensional maps of the environment. Typical cameras can provide 30 or more images per second and each pair of images can provide an almost complete map of the environment[1]. However, processsing even small low resolution (200×200 pixel) images in software takes more than a second in software - ignoring any post-processing needed to determine the velocity of an approaching object and the optimum strategy for avoiding it. This is well below the frame rates obtainable with commodity cameras and may be far too slow to enable even relatively slow moving objects to avoid each other. Thus hardware accelerators are required in order to obtain real-time 3D environment maps. Software simulations have determined that correlation techniques using a simple sum of absolute differences (SAD) cost function perform well[1,2]. Higher quality matching can be obtained with graph cut algorithms[3], but the processing time is two orders of magnitude longer and the algorithm lacks the regularity needed for efficient hardware implementation[4]. Area-based correlation algorithms attempt to find the best match between a window of pixels in one image and a window in the other. Matching is simplified if the system is aligned so as to meet the epipolar constraint - implying that matching pixels must be found in the same scan line in both images. The matching process is illustrated in Figure 1 which shows how a right image window is shifted by an amount known as the disparity until the best match is found with the reference window in the left image. In the SAD algorithm, the criterion for the best match is minimization of the sum of the absolute differences of corresponding pixels in a window, w:

$$C(x,y,\delta) = \sum_{x,y \subset w} |I_L(x,y) - I_R(x-\delta,y)$$

$C(x,y,\delta)$ is evaluated for all possible values of the disparity, δ, and the minimum chosen. For parallel camera axes, δ ranges from 0 for objects at infinity to Δ, corresponding to the closest possible approach to the camera. In collision avoid-

[1] Whilst complete maps are, in general, not attainable because some parts of the environment are not visible to both cameras simultaneously, this does not present a significant problem for the collision avoidance application.

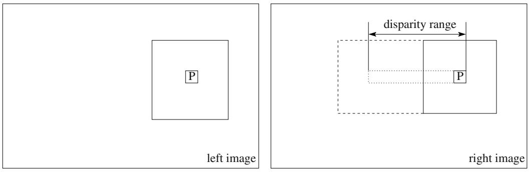


Fig. 1. Correlation based matching: the window centred on pixel, P, in the right image is moved through the disparity range until the best match (correlation) is found with a window centred at P in the left image. Aligning the cameras to meet the epipolar constraint ensures that P must lie on the same scan line in each image.

ance applications, Δ is readily set by considering the closest safe approach of the vehicle to an object. Correlation algorithms have regular structures and simple data flow making them excellent candidates for implementation in reconfigurable hardware. They also have abundant parallelism: $C(x, y, \delta)$ can evaluated in parallel for each $\delta \in [0, \Delta]$. The cost function requires only adders and comparators for which modern FPGAs provide good support. However accurate depth maps require large disparity ranges and high resolution images - both of which provide challenges to fitting a full correlator on a single FPGA. The aim of the study presented in this paper was to provide a simple predictor which would determine whether any given application, with its associated accuracy, speed, field of view, *etc.*, requirements could be fitted onto a single commercially available FPGA.

1.1 Stereo Hardware

Woodfill and von Herzen claimed that the Census algorithm implemented on an FPGA could compute depth maps for 320 × 240 pixel images at 42 frames per second for $\Delta = 24$[5]. The Census transform simply orders corresponding pixels in the left and right images and uses a single bit to indicate either '$<$' or '$\geq$', producing a simple, fast circuit. However, its performance falls far below that of SAD (or other correlation algorithms) and it is not very robust to noise[6]. Modifying the original ordering relation improves performance slightly, but not enough to match SAD[7]. Recently it has been used by Plakas[8] for real-time videoconferencing, but this is not a critical application and low performance may be tolerable. In related videoconferencing work, Schreer *et al.* could produce 'acceptable' depth maps in real time on an 800MHz Pentium[9], but the only statistics they provide are 'accepted' disparities, so it is unclear how their pyramidal approach (which has error propagation problems) affects resolution accuracy. In contrast, collision avoidance is a critical application: even if a system is used merely to assist a human operator, accuracy and the absence of false alarms become important.

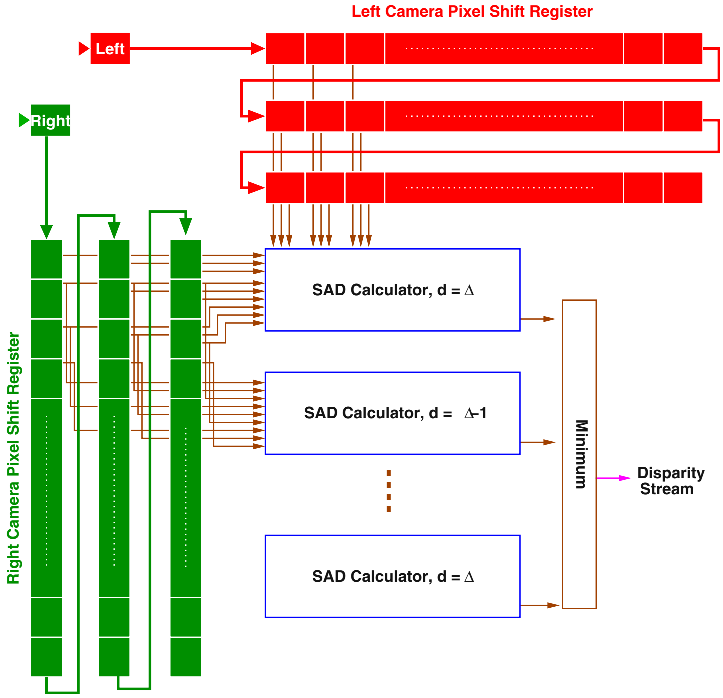


Fig. 2. Block diagram for SAD correlator: Note that the disparity calculator blocks are the source of parallelism in this circuit: each one is independent and operates in parallel.

2 Experiments

We have implemented SAD correlation algorithms in VHDL and synthesized them to determine their resource requirements and performance. Several approaches to handling high resolution images (with long scan lines which must be stored) have been investigated - storing scan lines in on-chip memory and altering the window shape to reduce the number of scan lines which must be stored.

2.1 SAD Correlator

A block diagram of the SAD circuit is shown in Figure 2. Pixels stream in from both cameras into the long left and right shift registers which store sufficient pixels so that all the pixels in a correlation window are available to the disparity calculators at the same time. The key parameters determing the size and performance of an SAD correlator are:

- *sl* - the scan line length,
- *wh* - the window height,

Table 1. SAD Correlator Resource requirements

Resource	Number required	Comments
SAD Correlator		
shift registers	2	Left and right registers are internally identical, but right register must provide more output connections
disparity calculators	$\Delta + 1$	Possible disparities range from $0..\Delta$
minimum detector	1	determine the minimum of $\Delta + 1$ sums
shift register size		
pixel registers	$(wh - 1)sl + ww\Delta$	
disparity calculator		
subtractors	$wh \times ww$	one subtractor for each pixel in a window
adders	$wh \times ww - 1$	correlator sum
minimum detector		
comparators	Δ	minimum of 2 (8-bit) pixels

– ww - the window width *and*
– Δ - the maximum disparity.

The basic resources have the following sizes:
Basic resource requirements are indicated in Table 1. To a first approximation, the resource requirements for an SAD correlator are given by:

$$\begin{aligned} cost_{SAD} \approx & (\Delta + 1)(c_{AD}wp + (wp - 1)c_{sum}) && \text{(disparity calculators)} \\ & + \Delta \cdot c_{comp} && \text{(minimum)} \\ & + (2(wh - 1)sl + ww(\Delta + 2) + 1)c_{reg} && \text{(shift registers)} \\ & + c_{overheads} && \text{(control, \textit{etc.})} \end{aligned} \tag{1}$$

where

$$\begin{aligned} wp = & \; wh \cdot ww = \text{number of pixels in matching window} \\ c_{AD} = & \; \text{cost of absolute difference circuit} \\ c_{sum} = & \; \text{cost of an adder} \\ c_{comp} = & \; \text{cost of a comparator} \\ c_{reg} = & \; \text{cost of a pixel register} \\ c_{overheads} = & \; \text{cost of control and steering logic} \end{aligned}$$

This relation should be a good predictor for low values of all the application parameters, where all overheads can be lumped effectively into the single overheads term. However, as packing density increases, poor placement would be expected to induce non-linear effects and to cause the overall cost to exceed that predicted by this model.

Key contributors to the delay of the correlator are the $wh \times ww - 1$ adders in each disparity calculator. A simple VHDL model which performs the additions in a loop adds a delay of $\mathcal{O}(wh \times ww)$ to the circuit. This sequential adder has two advantages:

1. the code in the VHDL model is trivial and
2. the sequential circuit generated[2] is very regular.

However, it is well known that a tree adder takes exactly the same number of circuit elements but improves the performance to $\mathcal{O}(\log ww \times wh)$[4]. The tree adder can be modelled in VHDL[10] and simulated. However, synthesizers tend to require 'statically determinable' numbers of circuit elements and are not prepared to run recursive code to build the tree. Whilst hand-coding adders for any particular window size is possible, stereo applications can have quite variable requirements and hand-coding models for many window sizes is not an attractive option. We overcame this problem by writing small programs in Python and Java[3] which generated the necessary VHDL code from a handful of input parameters[4]. The data in Table 2 shows the benefits of using the various adders. Two values for the tree adder are shown: one in which the model uses an 8-bit ripple-carry adder generated from full-adder blocks and the other in which the '+' operator is used. The latter model allows the synthesizer to produce more efficient packing of individual 1-bit adders (*cf.* the $\delta s/\delta p$ column) and uses the fast-carry logic to produce faster adders ($\delta t/\delta p$ column). Note that the synthesizer was able to produce a more compact circuit with the tree adder using the '+' operator, despite the triangular shape of the tree. Forcing the pixel adders to use small blocks of columns (so that they can use the fast carry logic) seems to have a good effect on overall routing, presumably because the placement module is constrained - with fewer degrees of freedom, it has less opportunity to produce a bad allocation of functions to logic blocks.

These results also show that a 'useful' circuit fits easily into a modern FPGA: commonly available small CMOS cameras have scan lines of the order of 200 or so pixels, $\Delta = 27$ implies that depths in the critical region may be measured to $\sim 4\%$ and a delay of $100ns$ allows a pixel clock of $10MHz$ - A 270×270 image at $30fps$ requires a $2.2MHz$ clock. Simulation shows that 9×9 windows produce good quality matching over a range of test images[2].

The last line in Table 2 shows an attempt to determine the largest value of Δ that the target chip could handle for the larger (9×9) window. At $\Delta = 33$, the model will fit, using 95% of the slices in a Xilinx XC2V8000 (8×10^6 'gates'). At

[2] This assumes that synthesizers are not yet able to recognize the potential to improve on the simple circuit as we have done by the procedure described next. We are optimistic that this will change soon!

[3] A consequence of the global distribution of the authors of this paper!

[4] Both versions were designed for re-use, so require the operator and operand type (*e.g.* bit width) as well as the circuit order.

Table 2. Effect of different adder styles

Adder type	Δ	Window size	Slices (Vertex2)	$\delta s/\delta p$	Progation delay (ns)	$\delta t/\delta p$
Sequential	27	3×3	8873	486	93.5	0.488
		9×9	43864		128.6	
Tree						
RC adder	27	3×3	9051	494	100.6	0.085
	27	9×9	44670		106.7	
+ operator	27	3×3	7363	421	100.3	0.066
	27	9×9	37695		105.1	
	33	9×9	46103[a]		105.1	

Pixel size: 8 bits; Scan line length: 270 pixels

[a] 95% of a Xilinx XC2V8000.

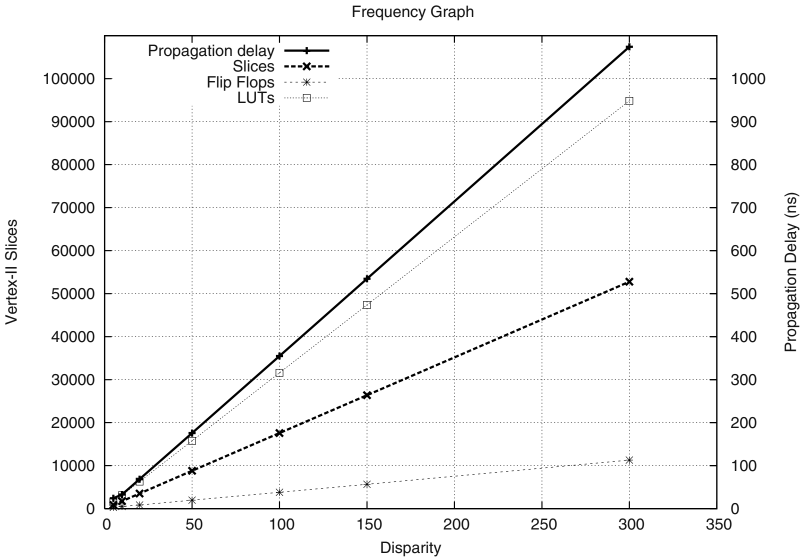


Fig. 3. Performance *vs* Δ. Pixel size: 8 bits; Window: 3×3; Scan line length: 270 pixels

this point, routing constraints start to dominate and although a simple linear model predicts that $\Delta = 34$ would also fit, there are insufficient routing resources and the router fails.

Accuracy depends on the disparity range that can be used, so we ran trials to determine the effect of increasing Δ on resource usage and performance. Figure 3 shows the regular structure of an SAD correlator leads to close to the expected linear relationship between Δ and logic cells needed.

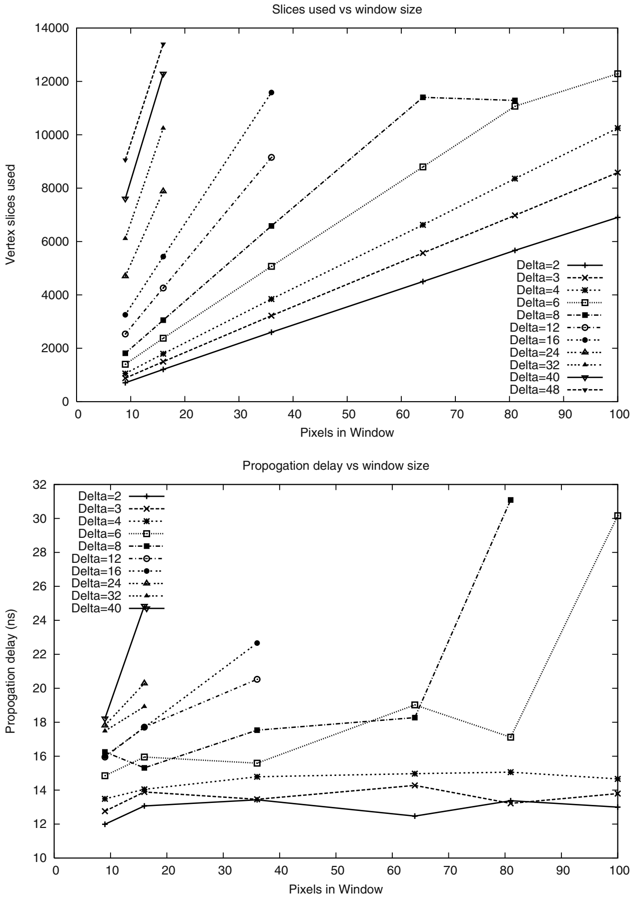


Fig. 4. Resource Usage *vs* window size for various Δ values

2.2 Using Block Memory

Typical FPGA logic cells contain look-up tables to implement the logic and a small number of flip-flops. Generally, these flip-flops provide adequate memory for the implementation of state machines, but the overall bit density is far too low for them to be used efficiently as 'plain' memory. To address this problem, modern FPGAs provide block memory - which can be configured in various ways, including as shift registers. The next set of trials sought to determine whether a practical circuit would fit onto a much smaller FPGA using the block memory to store scan lines. The family of curves in Figure 4 were obtained using the block memory on a Xilinx Vertex device to store the 'idle' (*i.e.* not involved in current matching) portions of scan lines. Results are presented for a much smaller ($\sim$ 10000 slices) suitable for an environment with cost or power constraints. A practical device is still feasible, but the disparity range (which

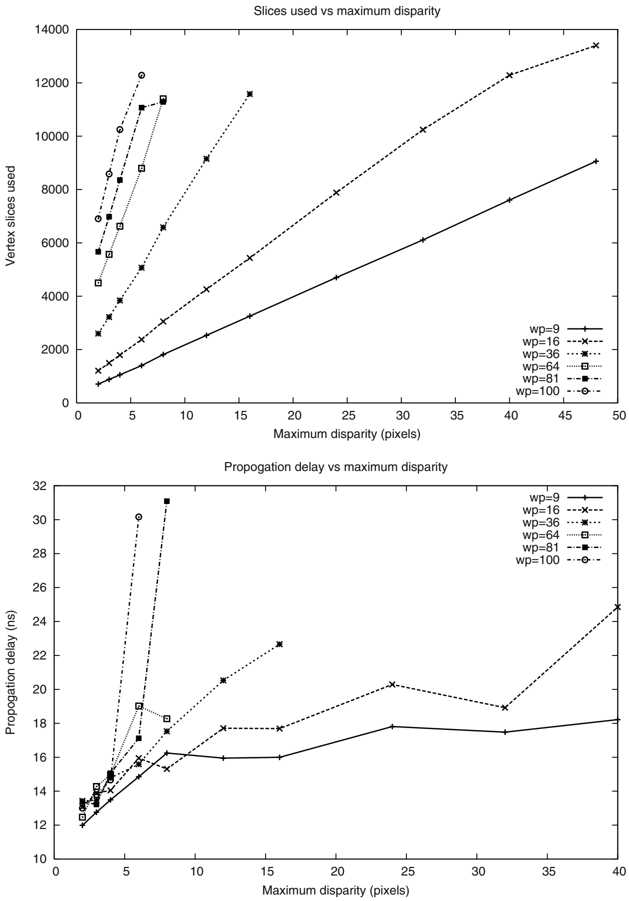


Fig. 5. Resource Usage *vs* Δ for various window sizes ($wp = ww \times wh$)

affects distance accuracy) and the window size (which affects matching quality) must be limited: but an 8×5 window (*cf.* Section 2.3) with $\Delta = 15$ is feasible. Real-time performance is easily achieved - see Figure 5(b) - without the need for pipelining. Accuracy and matching quality can be improved by using time series images[11,12] so it may be more important to achieve real-time performance than good initial matching.

2.3 Rectangular Windows

It is customary to use square matching windows in correlation-style stereo algorithms, probably because this is the simplest approach in a software system and there is little performance or other penalty. The matching process is generally only using a small part of each scan line at any time - specifically ww from the left image and $\Delta+1+ww$ from the right image. The remaining pixels are stored

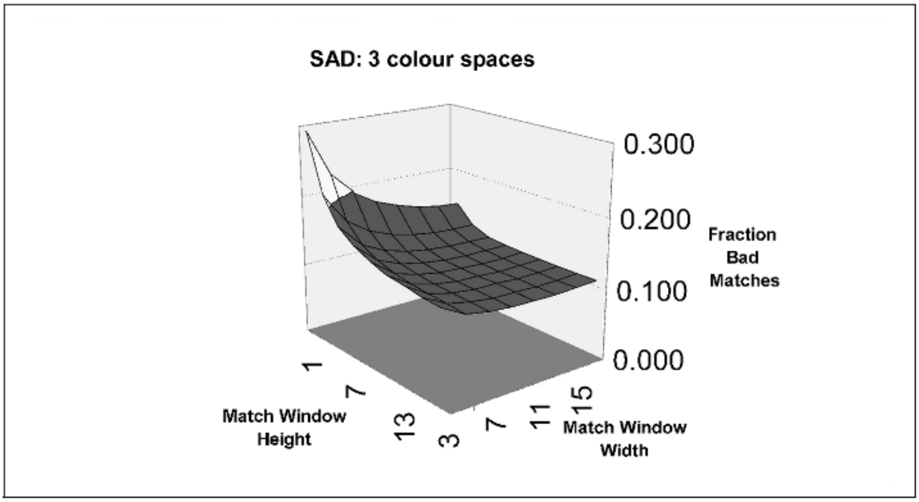


Fig. 6. Matching with rectangular windows: ‘Corridor’ data set[13]

in shift registers for use in subsequent cycles. In a simple model, these shift registers are trivially implemented in VHDL and synthesized to use the flip-flops in logic cells. Once the epipolar constraint has been satisfied, matches for each pixel should be found in the same scan line. Pixels in surrounding scan lines are only used to support matching by reducing noise effects. We ran a set of experiments to determine the effect of using different shaped matching windows: Figure 6 shows a large flat region with $\sim 10\%$ bad matches. Detailed examination of the actual numbers shows that a ‘short’ (low wh) wide window produces similar matching quality to a taller square one, *i.e.* the total number of pixels in the matching window is the critical factor. This implies that a considerable amount of space can be saved in an FPGA by using rectangular ($wh < ww$) windows without sacrificing matching quality.

3 Estimating Resource Needs

It is common for papers demonstrating the ‘success’ of some application to simply state a claim for that particular application, with little attention to extensions and variations. For stereo applications, this is particularly frustrating because every application presents its own criteria for accuracy, reliability, resources required, *etc.*Hence this work focussed on using the data gathered to enable feasibility of a proposed application to be quickly estimated. Whilst this can, in principle, be done by simply counting circuit elements needed to implement a module and using those counts in Equation 1, a place and route tool has to work from high level models and may have problems allocating and laying out circuits that a human engineer may not. FPGA implementations are also constrained by availability of routing resources and this factor is much harder to estimate than

Table 3. Cost factors

Cost	Range	Notes
Disparity calculator		
$c_{AD} + c_{sum}$	16-20	Essentially 2 8-bit adders The additional data needed to distinguish between c_{AD} and c_{sum} doesn't justify the small benefit - *cf.* Equation 1.
Minimum		
c_{comp}	14	8-bit comparator + steering logic
Shift Registers		
Using logic cells	7.3	1 eight bit pixel
Overheads		
$c_{overheads}$	0	Small - current data is insufficient to provide a reliable estimate

logic cell needs, thus practical trials of the type we carried out here are needed to determine real cost factors.

4 Conclusion

Accurate real-time 3D depth maps are feasible with modern FPGA technology. Simulations of the effect of changing the window shape show that altering the shape of the correlation window can be used to reduce the number of cells needed for 'inactive' parts of scan lines. Since even a 200 pixel scan line uses space similar to that required for a disparity calculator (working on a window large enough to produce good matching quality), it is clearly better to use the space to increase accuracy with more disparity calculators. Transferring the 'inactive' parts of scan lines to on chip memory enables small real-time systems to be implemented in quite small (*i.e.* 20% of current state-of-the-art) and thus economic systems.
The simple (unpipelined) circuits described here will provide real-time performance at pixel clock rates up to $\sim 10MHz$. For higher resolution images (and thus higher pixel clock frequencies if the same frame rate is to be maintained), an SAD correlator is easily pipelined. Three modules (SAD calculation, adder and minimum detector) of are readily identified in the data path suggesting that pixel clock rates of $\sim 2-3$ times greater (*i.e.* $\sim 20-30MHz$) are easily attained. The tree adder is the slowest of these and additional pipeline stages may be easily inserted within it. For large Δ circuits, the minimum detector will become the slowest element, but it is easily pipelined in the same way as the adder. In fact, since the major calculation elements are mainly 8-bit adders, the propagation delay for an 8-bit adder (plus pipeline overheads!) represents the lower limit for pixel clock time.

Acknowledgements. John Morris was supported by the Foreign Professors Invitation Program of The Korean IT Industry Promotion Agency at Chung Ang University in 2003-4.

References

1. Scharstein, D., Szeliski, R.: A taxonomy and evaluation of dense two-frame stereo correspondence algorithms. International Journal of Computer Vision **47** (2002) 7–42
2. Leclercq, P., Morris, J.: Assessing stereo algorithm accuracy. In Kenwright, D., ed.: Proceedings of Image and Vision Computing'02. (2002) 89–93
3. Boykov, Y., Veksler, O., Zabih, R.: Fast approximate energy minimization via graph cuts. In: ICCV 1999. (1999)
4. Morris, J.: Reconfigurable computing. In Oklobdzija, V.G., ed.: Computer Engineering Handbook, CRC Press, CRC Press (2001) 37–1 – 37–16
5. Woodfill, J., Herzen, B.V.: Real-time stereo vision on the PARTS reconfigurable computer. In Arnold, J., Pocek, K.L., eds.: Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines, Napa, CA (1997) 201–210
6. Leclercq, P., Morris, J.: Robustness to noise of stereo matching. In: Proceedings of International Conference on Image Analysis and Processing'02. (2003) 89–93
7. Leclercq, P.: Evaluation of Stereo Matching Algorithm for Hardware Implementation. PhD thesis, University of Western Australia (2004)
8. Plakas, C., Trucco, E., Brandenburg, N., Kauff, P., Karl, M., Schreer, O.: Real-time disparity maps for immersive 3-d teleconferencing by hybrid recursive matching and census transform. In: VideoRegister01. (2001) xx–yy
9. Oliver Schreer, N.B., Kauff, P.: Real-time disparity analysis for applications in immersive tele-conference scenarios - a comparative study. In: ICIAP2001. (2001) x–x+5
10. Ashenden, P.J.: Recursive and repetitive hardware models in VHDL. Technical Report TR160/12/93/ECE, Electrical and Computer Engineering, University of Cincinatti, Cincinatti, Ohio 45221-0030 (1993)
11. Torreao, J.R.A.: Estimating 3D shape from the optical flow of photometric stereo images. Lecture Notes in Computer Science **1484** (1998) 253–??
12. Barniv, Y.: Error analysis of combined stereo/optical-flow passive ranging. In: SPIE Proceedings. Volume 1479., NASA/Ames Research Ctr., Moffett Field, CA, USA (1991) 259–267
13. Gerdes, V.: Modular Rendering Tools. `www-student.informatik.uni-bonn.de/~gerdes/MRTStereo/index.html` (2001)

A Router Architecture for QoS Capable Clusters[*]

Madhusudhanan Anantha and Bose Bella

School of Electrical Engineering and Computer Science, Oregon State University,
Corvallis, Oregon 97331
madhusan@cs.orst.edu, bose@eecs.orst.edu

Abstract. Many web servers and database servers make efficient use of clustering from cost, scalability and availability standpoints. Existing cluster interconnects use switching schemes which minimize transmission latency but do not provide any guarantee on the delay (e.g., Wormhole switching). The variety of applications which are run on clusters mandate that the cluster interconnect be capable of handling both best effort and delay bound traffic. A new router architecture capable of providing **soft guaranteed** service using wormhole switching was proposed. An improved router model with preemption capabilities has also been proposed in literature. In this paper a detailed analysis of the hardware complexity of the preemptive router is presented and some architectural modifications to reduce the hardware complexity of the preemptive router are proposed. An interconnection network simulator has been developed to compare and analyze the performance characteristics of the proposed router architecture.

1 Introduction

Commodity clusters have a high performance to cost ratio over commercial parallel computers. This has lead to their use in a lot of applications which require high processing power. These include applications which they were not intended for e.g., web-servers. The increase in multi-media applications has meant that web-servers should be capable of handling time sensisitive traffic. Since existing switching schemes do not provide a guaranteed delay and since the cost of providing different switching schemes for different traffic types using the network is high we try to use one switching scheme with some modifications to handle all traffic types. A new router architecture capable of providing such service using wormhole switching with a rate-based scheduler was proposed in the MediaWorm router design [8] . A preemptive router architecture, which can dynamically allocate a Virtual Channel(VC) to any traffic class, was proposed by Das et.al. [1]. In this paper the hardware complexity of the premptive router is analyzed and some architectural modifications to reduce the hardware complexity of the router without sacrificing performance are proposed. An interconnection network

[*] This work is supported by National Science Foundation Grant CCR-0105204

P.-C. Yew and J. Xue (Eds.): ACSAC 2004, LNCS 3189, pp. 321–334, 2004.

simulator has also been developed to compare the performance characteristics of the proposed router with the preemptive router.

In Sect.2 some background work in this area is presented. In Sect.3 we analyze the hardware complexity of the router presented in Sect.2. Section 4 describes the modifications to the router architecture. In Sect.5 we present an analytical model to estimate the performance of the proposed router. Then, in Sect.6, we describe the simulation frame work and compare the performance characteristics of the modified router with the existing router. The conclusions and future work are summarized in Sect.7.

2 Background

2.1 Baseline Router Model

The most intuitive way to support QoS provisioning, in wormhole switched networks, is to attach a notion of priority to the traffic flows and add a rate based scheduling scheme like Virtual Clock [4]. This allows us to provide soft guaranteed service to time bound data streams with minimal changes to the wormhole router hardware.

Non-preemptive routers statically divide the VCs among the traffic classes. This restricts the flexibility of the router to handle changes in the distibution of traffic priority classes. The proposed solution to this problem is to use a preemptive router model which allows several priority classes of traffic to share the same VC, with the provision that higher priority message can preempt a lower priority message. The preemptive model can dynamically handle changes in traffic better than the non-preemptive model and hence is well suited for this application. A pipelined router model with the aforesaid capabilites was proposed in [8].

The modifications to this router model suggested in [1] are:

1. **Preemption in the input buffer**: Preemption in the input buffer occurs when the header flit of a high priority message arrives at the input VC buffer, which is being held by a lower priority message. When this happens the router lets the higher priority flow use the buffer and resumes transmission of the lower priority message after the higher priority message releases the buffer.
2. **flit acceleration mechanism**: The next modification they proposed was the use of a flit accelerate mechanism to reduce the probability of a higher priority message getting struck at the later pipeline stages due to the presence of lower priority flows. Flit acceleration mechanism solved this problem by boosting the priority of the lower priority flow holding the resource which is needed by the high priority flow so that the blocking delay of the high priority flow is minimized.

3 Hardware Complexity Analysis

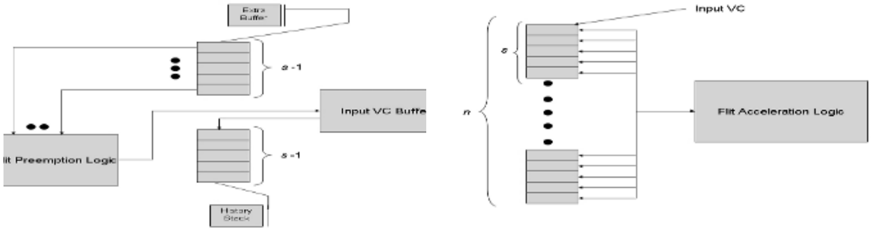
The architectural modifications proposed in Sect.2 incur a cost in implementation. In this section the harware complexity analysis of the *flit preemption logic* and the *flit acceleration logic* is presented. This analysis is used to investigate the possibility of modifications to the architecture to reduce the hardware complexity without sacrificing performance. Throughout the analysis n is the number of dimensions in the interconnect and s is the number of prioritized flow classes. The complexity is expressed in terms of number of gates used for a typical implementation.

3.1 Cost Analysis of the Flit Preemption Unit

Figure 1 shows a schematic of the flit preemption unit. The flit preemption unit has to do the following tasks in a clock cycle.

1. check the extra buffer to see if there is a header flit.
2. if there is a header flit, check whether this flit can preempt a flow in the virtual channel buffer.
3. if the tail flit of the preempted flow has already been received in which case a dummy tail flit is not created.
4. it is clear that at any given clock cyle there is only one buffer with a header flit in the extra buffer that needs to be processed.

It can be seen from Fig.1 the hardware complexity of the extra buffer and history stack scale in the order of number of prioritized flow classes at each input port. The hardware complexity of the flit preemption logic scales in the order number of VCs per physical channel because it has to maintain status information of all the VCs in the input buffer. However, this is fixed and its hardware complexity is constant. The effective hardware complexity of the flit preemption unit is:



(a)Flit Preemption Unit (b) Flit Acceleration Unit

Fig. 1. Modifications to Pipelined Router Model

- *flit preemption logic* = c
- *extra buffer complexity* = $(n \times s)+c$
- *history stack complexity* = $(n \times s)+c$
- *flit preemption unit complexity* = $2(n \times s)+3.c = O(s)$ (because $s \geq n$)

where c is a constant.

3.2 Cost Analysis of the Flit Acceleration Unit

Figure 1 shows a schematic of the flit acceleration unit. The flit acceleration unit has to do the following.

1. Check to see if any lower priority flow is occupying an output buffer. There are $(n \times s) - 1$ possible channels that have to be checked for a given channel.
2. if there exists such a buffer then the accelerate flag is set in the buffer.

The effective complexity in the flit acceleration unit is in the *flit acceleration logic*. Therefore it will suffice to compute the hardware complexity of the flit acceleration logic. This unit must check $(n \times s)$- 1 other flows to check if any of those flows is holding a resouce required by a given flow. This has to to done for all the $(n \times s)$ flows. Thus the hardware complexity of this unit is:

flit unit complexity = $(n \times s)(n \times s) = \mathrm{O}(s^2)$(because $s \geq n$)

Therefore it can seen that the effective complexity of the flit acceleration unit is higher than that of the flit preemption unit. In Sect.4 some modifications to the router to reduce the hardware complexity are proposed.

4 Proposed Modifications

From Sect.3 it can be seen that hardware complexity of the flit preemption unit is *linear* in the number of prioritized flow classes but the hardware complexity of the flit acceleration unit is *quadratic* in the number of prioritized flow classes. In this section we will propose modifications to the router architecture to replace the flit acceleration logic with lesser complexity functional units which perform the same function. These modifications are at the final stage in the router which multiplexes flits from various output VCs onto the phyiscal channel.

4.1 Buffer Status Aware Link Scheduling

The operation of an asynchronous flow control protocol between two routers is as follows:

1. Router 1 makes the request(RQ) line high to request permission to transmit a flit.
2. Router 2 sets the acknowledge(ACK) line to represent buffer availability and hence permission to transmit.
3. Router 1 begins transmission if the ACK line has been set to allow transmission of a flit. In which case the flit is transmitted as a series of *phits*.

A priority based link scheduler always tries to pick flits from the highest priority flow without considering the buffer occupancy status in the next router's input buffer. The disadvantage of this approach is that the router might keep picking a flit from a high priority flow for which there is no buffer space in the subsequent router repeatedly thereby wasting bandwidth and increasing the delay for other flows. The availability of buffer space for a flow in the subsequent router can be predicted from the value of acknowledgement received for the last sent flit. Therefore, it would be useful to include this information to decide which flow's flits must be picked for transmission. A single bit called ACK status($1 =$ buffer space available, $0 =$ no space) is associated with each VC in the output buffer. The scheduler picks a flit using both this bit and the priority of the flow in the buffer. Link scheduling algorithms which use both of these pieces of information are proposed.

Highest Non N-Ack Flow.

1. Get the list of channels ready for transmission at a given clock cycle. Let $\{S\}$ be the set of all channels which can be scheduled and $\{R\}$ be the set of ready channels. Assign $\{S\} \leftarrow \{R\}$
2. Pick a channel from $\{R\}$ which has the
   (1) *Highest* Flow class.
   (2) Does not have an *N-Ack* (Negative Acknowledgement).
   (3) Choose the *first such channel* in case of a tie.
3. Read the flit from this channel and record the ACK status for the next iteration.

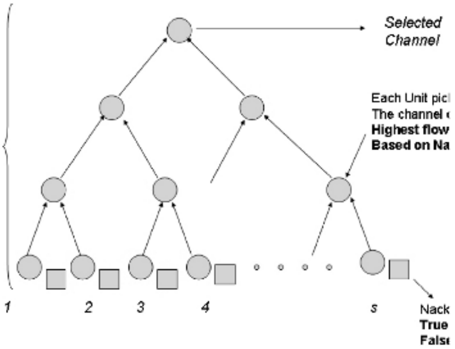


Fig. 2. Hardware implementing Highest Non N-ACK Scheduling

Discussion: Figure 2 shows a hardware implementation of the link scheduling algorithm.The algorithm uses a tree shaped circuit to pick the "winner" channel at a given clock when its starts flit transmission. At each clock cycle a flit from the selected channel is transmitted and the ACK status of the transmission is stored for use in a subsequent transmission.

Note that only one bit is added to the lowest level processing units because this level “filters” all channels that are ready and the subsequent levels work by picking a flow to *highest priority* among the ready channels. Thus the only addition in hardware complexity is the *O(s)* bits.

The **disadvantage** of this algorithm is that a high priority flow class which received an N-ACK might not be picked for a long time if there are enough flits from other flows. In essence, a high priority flow might “starve”. This disadvantage becomes crucial in heavily loaded networks.

A simple fix to this problem lies in the introduction of new variables.

1. *cycle_wait* which keeps track of the number of cycles after receiving a N-ACK and is updated each clock cycle.
2. *max_count*, maximum value after which *cycle_wait* update should stop.
3. *cycle_wait* is *initialized/reset* after receiving a N-ACK after *max_count* was reached and a re-transmission was tried.

Modified Highest Non N-Ack Flow.

1. Get the list of channels ready for transmission at a given clock cycle. Let $\{S\}$ be the set of all channels which can be scheduled and $\{R\}$ be the set of ready channels; Assign $\{S\} \leftarrow \{R\}$
2. Pick the channel of the *Highest Flow class*, among $\{R\}$, that does not have a N-ACK or whose counter has reached *max_count*.
3. Choose the *first such channel* in case of a tie.
4. Read the flit from this channel and record the ACK status for the next iteration.

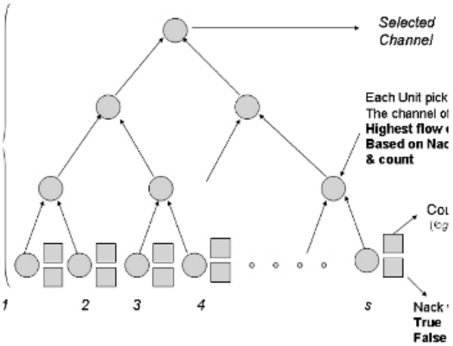


Fig. 3. Hardware implementing the Modified Highest Non N-ACK Scheduling

Discussion: Figure 3 shows the hardware implementation of the modified version of the algorithm. At each clock cycle a flit from the selected channel is transmitted and the ACK status of the transmission is stored for use in a subsequent transmission.

Note that one counter and a register of size *log(s)* are added to the lowest level processing units because this level "filters" all channels that are ready and have the required counter constraints. The subsequent levels work by picking a flow of the *highest priority* among the channels selected at level 1. Thus, the only addition in hardware complexity is the *O(s.log(s))* bits.

4.2 Flexible Output Virtual Channel Allocation

In this section we discuss the next modification to the router. The main disadvantage of using the link scheduling algorithms we presented is that even though they guarantee the presence of a free channel at the output VC buffer the fixed connection scheme between the crossbar ports and the VCs could lead to performance losses. This is because certain combinations of connections are not possible and header still incurs a blocking delay. To remedy this situation a dynamic mapping scheme between a crossbar port to a VC buffer is proposed.

Figure 4 shows a schematic of the proposed modification. The hardware addition to the router is a *channel identifier* at each crossbar output port which contains the address of the channel allocated to this port. Each output port is allocated a specific set of channels it can choose from and the size of this identifier is $\log(\log s)$ and there are multiplexer circuits of complexity $\log s$ giving an overall gate complexity of $s.\log s$.

4.3 Analysis

Figure 4 shows the router architecture with both of the modifications in place. These changes are valid in the context of replacing the *flit acceleration unit* because the main reason for introducing acceleration is to avoid the *blocking of high priority flows* due to other lower priority flows. Since the link scheduling algorithms work by maximizing the number of flits transmitted in a time frame and since the probability of servicing high priority flows is high, the probability

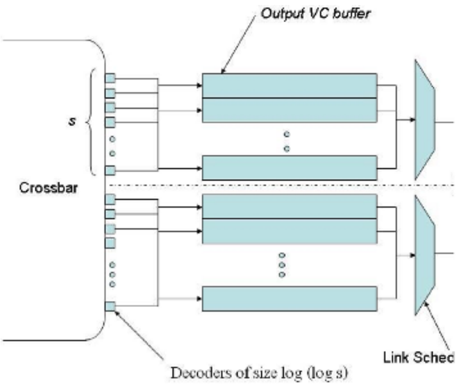


Fig. 4. Modified Router Architecture

of availability of a virtual channel is high. With a flexible channel allocation scheme to take advantage of the high availability of free channels the probability of blocking of a high priority flit is minimized. Therefore, we can see that this combination of modifications replaces the flit acceleration unit functionally.

The effective hardware complexity of the modifications is the total of the hardware complexities of the proposed modifications.

Total Complexity $= O(\text{s.log(s)}) + O(\text{s.log(s)}) \implies O(\text{s.log(s)})$ This hardware complexity bound is therefore better than that of the flit acceleration unit $(O(\text{s}^2))$ that has been replaced.

5 An Analytical Model

In this section a mathematical model for deadline missing probability in a single router is proposed. It is extended to calculate the deadline missing probability of a message which traverses r routers to its destination. A router model with a pipelined architecture with 5 stages and augmented with the modifications proposed in Sect.4 is assumed. Stage 1 of the router performs flit de-multiplexing. Stages 2 and 3 are the routing and arbitration units. Stage 4 contains the crossbar and stage 5 contains multiplexes VCs over the physical channel. The model is described for S classes consisting of $(S-1)$ classes of real time traffic and one class of best-effort traffic.

Note that a message entering the pipelined router can experience delay at stages 1, 3 and 5. If the input VC buffer is full in stage 1, the message must wait outside the router until adequate space is available. In stage 3, the message again might be delayed because the destination crossbar ports are full. Crossbar output port arbitration is performed at a message level granularity. So the message has to wait until the required port is released by the message already holding it. Finally in stage 5, multiple virtual channels compete for the physical channel bandwidth. This is the delay experienced due to the link scheduler. The model is based on the following assumptions:

1. The arrival pattern of each class s follows a poisson process with an average arrival rate λ_s^g .
2. Message length is M flits long.
3. Message destination is uniformly distributed.
4. The input and output virtual channel buffers in stages 1 and 5 can hold b_s flits.
5. Each class is assigned separate injection/ejection queues outside the router, and these have infinite capacity.

The average message latency of a message of class s $(1 \leq s \leq S)$ is composed of the average network latency, $\overline{L_s}$, which is the time to traverse the router (network), and the average waiting time, $\overline{W_s}$, at the injection channel. Thus,

$$\overline{Latency}_s = \overline{L_s} + \overline{W_s} \tag{1}$$

Network latency can be calculated as follows

$$\overline{L_s} = P - 1 + (M + \overline{B_s})\overline{S_s} \tag{2}$$

where $\overline{L_s}$ is the network latency of a class s message, $\overline{B_s}$ is the blocking length and $\overline{S_s}$ is the effect of the bandwidth sharing mechanism.

Blocking occurs in stages 1,3 and 5 and the average blocking length can be separated into three parts as

$$\begin{align} Input &= P[\text{input buffer is occupied}].\{M/2\} \tag{3}\\ Arbiter &= P[\text{Arbiter is busy}].\{M/2\} \tag{4}\\ LC &= P[\text{ouput buffer is full}].\{M/2\} \tag{5} \end{align}$$

where *Input*, *Arbiter* and *LC* represent the corresponding blocking lengths at stages 1,3 and 5. In (3), (4) and (5), the first term represents the probability that the corresponding buffer is not empty, and the second term is the average message length that will be affected due to blocking. For example, if the input buffer is not empty, the header flit will face an average delay of $M/2$ flits.

In order to calculate the average blocking length the probability that the input buffer is full, the probability that the output buffer is full, and the delay due to bandwidth sharing have to be calculated. These terms can be calculated as follows:

5.1 Average Blocking Length in Stage 1

The router uses a *preemptive model* of virtual channel allocation. The preemptive model can dynamically allocate any virtual channel to any traffic class. Assuming a buffer size of b_s, a flit is blocked if the input virtual channel buffer is full and there are no flows whose priority is lower than this flit. In this case the flit will have to wait for channel to be released and hence a delay is incurred. Therefore the blocking delay is

$$\begin{align} delay1_s &= P[\text{buffer is full with no flows priority} < s].\{M/2\} \tag{6}\\ P_s[\text{full}] &= P[\text{each channel has a flow} \geq s] \tag{7}\\ &= \prod_{i=0}^{S-1}\sum_{j=s}^{S-1} P[X_j] \tag{8} \end{align}$$

where,
$P_s[\text{full}]$ = P[buffer is full with no flows priority $< s$]
and $P[X_j]$ = probability that a message is of flow class j.

5.2 Average Blocking Length in Stage 3

The header flit is blocked waiting at stage 3 if the arbiter has higher priority flows in the arbitration slots which reserve all crossbar ports that this flow might require or if all the crossbar ports are occupied. The blocking delay is:

$$delay2_s = P[\text{Arbiter Busy}].\{M/2\} \tag{9}$$

$$P[\text{arbiter busy}] = \prod_{i=0}^{k} \sum_{j=s}^{S-1} P[X_j] + \left(1 - \sum_{i=1}^{l} {}^{l}C_i.p^i.q^{l-i}\right) \tag{10}$$

where M is the message length in flits, k is the number of arbitration slots, l is the number of crossbar ports that could be assigned, p is the probability that a given output VC is full and $q = 1 - p$.

5.3 Average Blocking Length in Stage 5

Since a flexible virtual channel allocation scheme is used a flit is blocked waiting at the last stage only if no virtual channels are empty in the output virtual channel buffer.

$$delay3_s = P[\text{Ouput Buffer full}].\{M/2\} \tag{11}$$

$$P[\text{no VCs}] = 1 - P[\text{at least 1 free channel}] \tag{12}$$

$$P[\text{no VCs}] = 1 - \sum_{i=1}^{S-1} {}^{(S-1)}C_i.p^i.q^{n-i} \tag{13}$$

where M is the message length in flits and n is the number of virtual channels in the VC buffer.

5.4 Deadline Missing Probability

Using the average blocking lengths at stages 1, 3 and 5 of the router pipeline the probability of a packet of class s missing a deadline D_s can be calculated. The average blocking length ($delay$) for a message in a router is:

$$delay_s = delay1_s + delay2_s + delay3_s \tag{14}$$

Given a delay ($delay_s$) the actual time for transfer (β_s) of the message is given by:

$$\beta_s = delay_s.S_s \tag{15}$$

where S_s is the average number of cycles to transmit a flit of class s.

Equation (14) gives the average delay for a message of class s through a *single* router. The probability that the message misses its deadline is given by:

$$P_{m,s}(D_s,\beta_s) = P\{\beta_s > D_s\} = 1 - P\{\beta_s \leq D_s\} = \sum_{i=0}^{i=B_l} P_s(i) \tag{16}$$

where $P_{m,s}$ is the probability of missing the deadline D_s for a class s message and β_s is the actual delay and $P\{\beta_s \leq D_s\}$ is the probability that a class s message traverses the router within the deadline D_s and B_l is the highest blocking length such that $\beta_s \leq D_s$.

If a message were to traverse r routers to its destination, the probability that it misses its deadline can be calculated as the sum of the probabilities of all combinations of delays at these routers such that the total delay is less than B_m (B_m is the maximum total length such that the message reaches on or before its deadline(D_s)).

$$P\{\beta_s > D_s\} = 1 - P\{\beta_s \leq D_s\} = \sum P_s(D_0).P_s(D_1)\cdots P_s(D_{r-1}) \qquad (17)$$

where

$$\forall D_0, D_1, \cdots, D_{r-1} \in \mathbb{Z}^+ \qquad (18)$$

$$D_0 + D_1 + \cdots + D_{r-1} \leq B_m \qquad (19)$$

Equation (17) represents all possible combinations of delays at the routers. We need a solution to $P_s(B)$ to calculate the deadline missing probability. Since it is tough to exactly calculate this parameter, we approximate this probability using operational behaviour of a router. Under the uniform distribution assumption this probability can be calculated as follows.

$$P_s(B) \approx \begin{cases} 1 - \sum_{i=1}^{B^u} P_{b,s}/B^u, \; B = 0 \\ P_{b,s}/B^u, & 1 \leq B \leq B^u \\ 0, & \text{otherwise} \end{cases} \qquad (20)$$

where, B^u represents the worst case blocking length at a router and $P_{b,s}$ is the blocking probability of a class s message.

The probability of blocking of a class s message can be calculated as follows.

$$P_{b,s} = \prod_{i=0}^{S-1}\sum_{j=s}^{S-1} P[X_j] + (1 - \sum_{i=1}^{S-1} {}^{(S-1)}C_i.p^i.q^{n-i}) + \prod_{i=0}^{k}\sum_{j=s}^{S-1} P[X_j] + (1 - \sum_{i=1}^{l} {}^{l}C_i.p^i.q^{l-i}) \qquad (21)$$

and worst case blocking length is given by

$$B^u = M + M + M \Rightarrow 3.M \qquad (22)$$

Therefore we can roughly estimate the deadline missing probability of a class s message through a series of routers using (17), (20), (21) and (22).

6 Performance Analysis

6.1 Simulation Framework

An interconnection network simulator has been developed to compare the performance characteristics of the modified router with the QoS enabled architecture discussed in Sect.2 and a router with the modifications we have proposed. For our experiments we simulated a 8-port router and a 2 × 2 mesh network with 8-port routers. We have used 16 VCs per physical channel. The flit size is 128-bits and all the messages are 36 flits long. Physical link bandwidth is 1.6 Gbps and the flit buffers are 36 flits deep.

6.2 Workload

The workload includes messages from real time variable bit rate (VBR) traffic and best-effort traffic. The real time traffic streams are generated as synthetic MPEG streams at 30 frames/sec with different bandwidth requirements. Each stream generates a frame of data which is fragmented into flits. Best effort traffic is generated with a given injection rate λ and follows a poisson distribution. The message destination is assumed to be uniformly distributed.

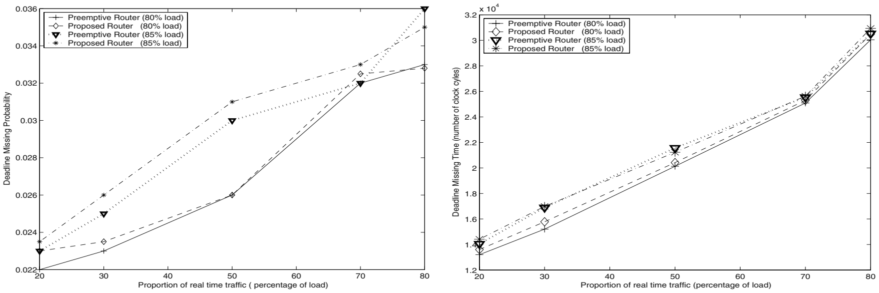
An important parameter that is varied is the input load which is expressed as a fraction of the physical link bandwidth. For a specific input load, we vary the ratio of the two classes ($x : y$, where $x/(x + y)$ is the fraction of load for VBR traffic and $y/(x + y)$ is the fraction of load for best effort component) to generate mixed-mode traffic.

We vary the traffic ratio in 5 stages during the simulation to simulate dynamic workload. The important output parameters we measured were the *deadline missing probability* and *deadline missing time*. Deadline missing probability is the ratio between the number of frames which missed their deadline out of the total delivered frames. Deadline missing time is the average time by which the packets miss their deadline. The deadline was set to 33.3 msec after receiving a frame from the flow. We have assumed a core clock frequency of 100 MHz and set the respective number of clock cycles for the deadline and the results are also expressed in terms of clock cycles.

6.3 Single Router Results

The simulation test bed was used to study the performance of the proposed router model with that of the existing router model. We ran the simulations at 80% and 85% network load. Figure 5.a shows the deadline missing probabilities of the proposed router model to the existing preemptive router model and Fig.5.b shows the deadline missing times of the two router models.

Figure 5.a shows that for both the routers the deadline missing probability increases as the proportion of real time traffic increases. It also shows that at 80% load the deadline missing probabilities of the models are close and the proposed router starts to perform better as the proportion of real time traffic increases.



(a) Deadline Missing Probability

(b) Deadline Missing Time

Fig. 5. Single Router Results

This is the phase where the advantages of the modified link scheduler start showing up as the scheduler tries to maximize the amount of data transmitted. At 85% network loading the deadline missing probabilities of the proposed router architecture and the existing router architecture are very close but the difference in performance starts increasing because of the modified link scheduler.

Figure 5.b shows the deadline missing times of the two router models at 80% and 85% loads are comparable. It also shows that the average deadline missing time increases as the proportion of real time traffic increases. It is also seen that the average deadline missing time increases as the ratio of real time traffic increases.

6.4 A (2 × 2) Mesh Network Results

Figure 6 shows that at 80% load and at 85% network loading the deadline missing probabilities of the proposed router architecture and the existing router architecture are very close. Like the single router results it is seen that the number of frames missing the deadline increases as the ratio of real time traffic increases.

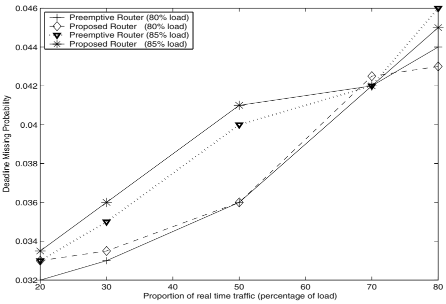


Fig. 6. Deadline Missing Probability (2×2 mesh)

From the results of the simulation it can be seen that the proposed router architecture achieves a slightly better performance than the existing model but at a reduced hardware complexity.

7 Conclusions and Future Work

This paper addresses the issue of hardware complexity in QoS capable routers to enable faster switching. The existing QoS capable router architecture was analyzed to identify sources of hardware complexity. Techniques like Buffer Status Aware Link Scheduling and Dynamic Output VC allocation were used to reduce the hardware complexity without sacrificing performance. Simulation was used to analyze the performance characteristics of the proposed router architecture. An analytical model for analyzing QoS capable clusters has been developed.

We are currently working on the verification of the proposed analytical model. It will be instructive to analyze the possibility of using adaptive routing and the preemptive router model in order to improve the QoS capabilities of the cluster interconnect.

References

[1] Das, C. R., Kim, E. J., and Yum, K. H.: QoS provisioning in clusters: An investigation of Router and NIC design. Proceedings of the 28th International Symposium on Computer Architecture, ISCA 01, Sweden, 2001

[2] Duato, J., Yalamanchili, S., and Ni, L.: Interconnection Networks: An Engineering Approach. Morgan Kaufmann Publishers, second edition, 2002.

[3] Das, C. R., Kim, E. J., and Yum, K. H.: Calculation of Deadline Missing Probability in a QoS Capable Cluster Interconnect. Proceedings of IEEE International Symposium on Network Computing and Applications (NCA '01), pp.34-43, February 2002, Cambridge, MA.

[4] Zhang, L.: VirtualClock: A New Traffic Control Algorithm for Packet-Switched Networks. ACM Transactions on Computer Systems, 9(2):101-124, May 1991.

[6] Chien, A. A.: A cost and speed model for k-ary n-cube wormhole routers. Proceedings of Hot Interconnects'93, August 1993.

[7] Duato, J., Yalamanchilli, S., Caminero, M. B., Love, D. and Quiles, F. J.: MMR: A High Performance Multimedia Router-Architecture and Design Tradeoffs. Proceedings of International Symposium on High Performance Computer Architecture. Pages 300-309, January 1999.

[8] Yum, K. H., Vaidya, A. S., Das, C. R. and Sivasubramaniam, A.: Investigation of QoS support for traffic mixes with the MediaWorm Router. Proceedings of International Symposium on High Performance Computer Architecture, pages 97-106, January 2000.

Optimal Scheduling Algorithms in WDM Optical Interconnects with Limited Range Wavelength Conversion Capability[*]

Zhenghao Zhang and Yuanyuan Yang

Department of Electrical & Computer Engineering, State University of New York, Stony Brook,
NY 11794, USA

Abstract. In this paper we study optimal scheduling algorithms to resolve output contentions in time slotted WDM optical interconnects with wavelength conversion ability. We consider the general case of limited range wavelength conversion with arbitrary conversion capability, as it is easier to implement and more cost effective than full range wavelength conversion, and also includes full range wavelength conversion as a special case. We consider the conversion scheme in which each wavelength can be converted to multiple wavelengths belongs to an interval and the intervals for different wavelengths are "ordered". To be specific, the conversion range of λ_i can be written as $[begin(i), end(i)]$, where $begin(i)$ and $end(i)$ are positive integers and if $i < j$ then $begin(i) \leq begin(j)$ and $end(i) \leq end(j)$. We will present linear time optimal scheduling algorithms for both buffered and unbuffered WDM switches. We will also give performance studies of these switches when scheduled by these algorithms.

Keywords: Wavelength-division-multiplexing (WDM), optical interconnects, scheduling, wavelength conversion, limited range wavelength conversion, bipartite graphs, bipartite matching, matroid.

1 Introduction and Background

All-optical networking with *wavelength-division-multiplexing (WDM)* is emerging as the candidate for future high-bandwidth communication networks. In this paper, we will study time slotted WDM packet switching networks as it may offer better flexibility and better exploitations of the bandwidth [9]. As in [9] [17], we assume that the duration of an optical packet is one time slot and the traffic pattern is unicast, i.e., each packet is destined to only one output fiber.

In a WDM switch, output contention occurs when more than packets on the same wavelength are destined to the same output fiber. To resolve output contention we can translate the wavelength of a packet to some other idle wavelength by *wavelength converters*. We consider *limited range* wavelength converter which is capable of converting a wavelength to a limited number of wavelengths, since it is more realistic and cost-effective to provide wavelength conversion ability than *full range* wavelength converter

[*] The research work was supported in part by the U.S. National Science Foundation under grant numbers CCR-0073085 and CCR-0207999.

P.-C. Yew and J. Xue (Eds.): ACSAC 2004, LNCS 3189, pp. 335–348, 2004.

which is capable of converting a wavelength to any other wavelengths [10,8]. Also, full range wavelength converters can be regarded as a special case of limited range wavelength converters.

Another way to resolve output contention is to temporarily store the contending packets into the buffers, as practiced in all electronic switches. However, it is difficult to directly apply it to WDM switches, since optical buffers are made of fiber delay lines and quite costly and bulky [2]. Therefore we will consider unbuffered WDM switches first. Later on we will also consider buffered WDM switches.

In these switches, scheduling algorithms are needed to smartly allocate the resources (the wavelength channels) to the requests (the arrived packets) to optimize network performance, such as network throughput. This problem was first realized by [17], in which a simple algorithm, "storing the packet in the least occupied buffer", was suggested. However the authors didn't show that the algorithm is optimal, in the sense that the network throughput achieves maximum and total delay achieves minimum. In this paper we will present algorithms that are optimal, for both buffered and unbuffered switches.

2 Wavelength Conversion

We make two assumptions about limited range wavelength conversion:

Assumption 1. *The wavelengths that can be converted to by λ_i for $i \in [1, k]$, where k is the number of wavelengths on a fiber, can be represented by interval $[begin(i), end(i)]$, where $begin(i)$ and $end(i)$ are positive integers in $[1, k]$. The wavelengths belong to this interval is called the* adjacency set *of λ_i.*

Assumption 2. *For two wavelengths λ_i and λ_j, if $i < j$, then $begin(i) \leq begin(j)$ and $end(i) \leq end(j)$.*

We call this type of wavelength conversion "ordered interval" because the adjacency set of an wavelength can be represented by an interval of integers, and the intervals for different wavelengths are "ordered". The cardinality of the adjacency set is called the *conversion degree* of the wavelength. Different wavelengths may have different conversion degrees. The conversion degree of the interconnect, denoted by D, is defined as the largest conversion degree of all wavelengths. The *conversion distance* of a wavelength is defined as the largest difference between a wavelength and a wavelength that can be converted from it.

We can use a bipartite graph to visualize the wavelength conversion. Let the left side vertices represent input wavelengths and the right side vertices represent output wavelengths. λ_i on the left and λ_j on the right are connected if λ_i can be converted to λ_j. Figure 1 shows such a conversion graph for $k = 8$. The adjacency set of λ_3, for example, can be represented as $[1, 5]$, and the conversion degree of λ_3 is 5.

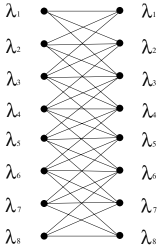


Fig. 1. Wavelength conversion in a 8-wavelength system with conversion distance 2.

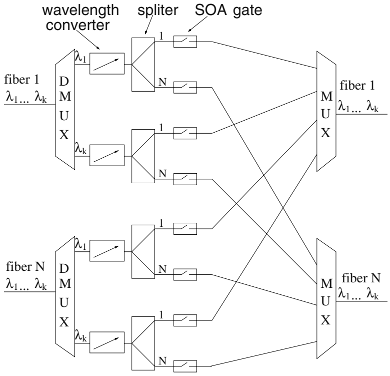

wavelength converter
spliter
SOA gate
fiber 1
$\lambda_1 \ldots \lambda_k$
DMUX
λ_1
λ_k
1
N
fiber N
$\lambda_1 \ldots \lambda_k$
MUX
fiber 1
$\lambda_1 \ldots \lambda_k$
fiber N
$\lambda_1 \ldots \lambda_k$

Fig. 2. A wavelength convertible WDM optical interconnect.

3 Optimal Scheduling in Unbuffered WDM Switching Networks

3.1 Network Model

An unbuffered WDM switch with wavelength conversion is shown in Fig.2. It has input N fibers and N output fibers. On each fiber there are k wavelengths that carry independent data. Thus, there are a total of Nk input wavelength channels and Nk output wavelength channels. It can be seen from the figure that an input fiber is first fed into a demultiplexer, where different wavelength channels are separated from one another. An input wavelength is then fed into a wavelength converter to be converted to a proper wavelength. The output of a wavelength converter is then split into N signals, which are connected to each of the output fibers under the control of N SOA gates. The signal can reach the output fiber if the SOA gate is on, otherwise it is blocked. Since the request has only one destination, only one of the SOA gates is in on at a time. In the front of each output fiber there is an optical combiner which multiplexes the signals on different wavelengths into one composite signal and send to the output fiber. Apparently, it is required that all signals to the optical combiner must be on different wavelengths.

To understand the problem that needs to be solved by the scheduling algorithm, we can use the following example. Consider a simple interconnect with 2 input/output fibers and 4 wavelengths per fiber shown in Fig.3. Suppose under limited range conversion, wavelength λ_i, $1 \leq i \leq 4$, can be converted to λ_j where $j \in [\max(i-1,1), \min(i+1,4)]$, as shown in the left part of the figure. At the beginning of a time slot, there are 4 packets on λ_1, λ_2, λ_3, λ_4 arrived at input fiber 1, destined for output fiber 2, 2, 1, 1, respectively. In the figure destination of a request is the number shown in the parenthesis. There are 2 packets on λ_1 and λ_2 arrived at input fiber 2, and all destined for output fiber 2. We first notice that there is no contention at output fiber 1, since there are only two requests destined to it, and they are on different wavelengths. These two requests can both be granted and no wavelength conversion is needed. However, there are contention at output fiber 2, since there are 4 requests, 2 on λ_1 and 2 on λ_2, destined for this output fiber. Without wavelength conversion, one request on each of the wavelengths must be dropped. With limited range wavelength conversion, 3 wavelengths, λ_1 to λ_3, can be converted from λ_1 and λ_2 and therefore 3 of the 4 requests destined for output fiber 2 can be granted. We can assign channel λ_1 to the request arrived at input fiber 1 on λ_1, assign channel λ_2 to the request arrived at input fiber 2 on λ_1, assign channel λ_3 to the request arrived at input fiber 1 on λ_2, and reject the request arrived at input fiber 2 on λ_2. Based on these decisions, the wavelength converters are configured to convert input wavelengths to proper output wavelengths, as shown in the figure. A SOA gate is set to on if the request is granted and is destined to the output fiber connected to the gate. We can see in the example that when contention arises at an output fiber, to maximize network throughput, we attempt to find the largest group of requests that are contention free.

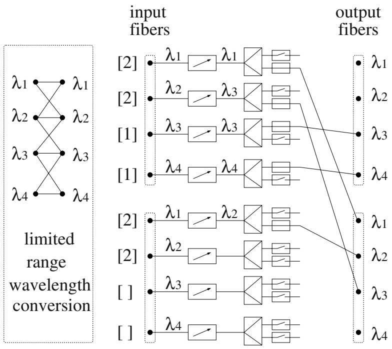


Fig. 3. Requests and wavelength channel assignments of an example interconnect with 2 input/output fibers and 4 wavelength per fiber. The number in the parenthesis are the destination of a request.

3.2 The First Available Algorithm

First, notice that if only to maximize network throughput, the packets destined to different output fibers can be scheduled independently, since a wavelength channel on output fiber p will not be assigned to a request destined to output fiber q if $p \neq q$. Therefore from now on we consider only one output fiber. The input to our scheduling algorithm will

Table 1. First Available Algorithm

```
current:=1;
for $i := 1$ to $k$ do
    Find a packet on a lowest wavelength
    convertible to $\lambda_i$ that has not been
    assigned to any wavelength channel yet.
    if there is such a packet
        assign $\lambda_i$ to this packet;
    end if
end for
```

be the packets destined to this fiber. The output of the algorithm will be the decisions about whether a packet is granted or not, and if granted, which wavelength channel it is assigned to. To maximize network throughput, the algorithm should be able to find the maximum number of packets that can be granted without causing contention for any possible input pattern. The algorithm can be run independently and in a distributed manner to speed up the scheduling process.

[13] showed that due to the properties of limited wavelength conversion, the optimal scheduling can be found by a simple algorithm called the First Available Algorithm shown in Table 1. This algorithm scans the wavelength channels on the output fiber from λ_1 to λ_k. At step i, λ_i, can be assigned to packet a if (1) a has not been assigned to any wavelength channel yet, (2) the wavelength of a can be converted to λ_i. Moreover, the key part of the algorithm is that, it will find such a packet on the lowest wavelength, or the “first available” one.

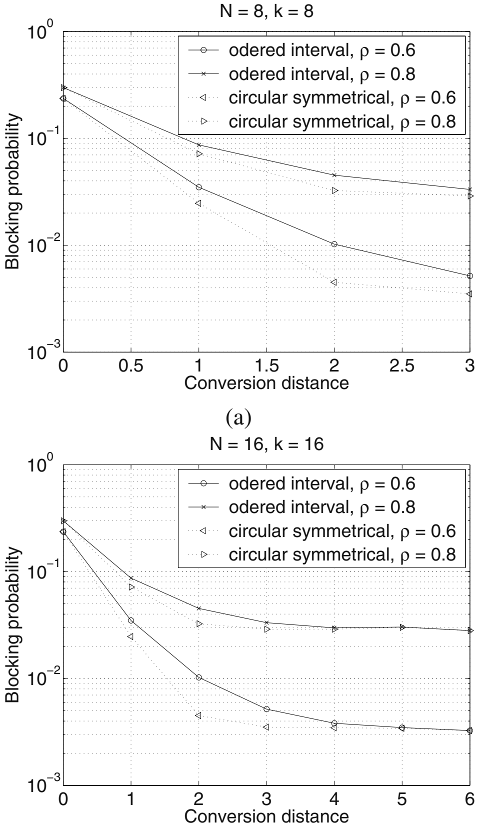
The complexity of the algorithm is $O(k)$ where k is the number of wavelengths on a fiber, since the loop is executed exactly k times and the work within the loop can be done in constant time. However, the scheduling time is not completely independent of network size N, since to generate the input to the algorithm one might have to scan all the input channels.

4 Simulations

We implemented the proposed algorithm in software and tested it by simulations. We tested the interconnects of two typical sizes, one with 8 input fibers and 8 output fibers and with 8 wavelengths on each fiber, and the other with 16 input fibers and 16 output fibers and with 16 wavelengths on each fiber.

In the simulations, we assume that the arrivals of the packets at the input channels are bursty: an input channel alternates between two states, the “busy” state and the “idle” state. When in the “busy” state, it continuously receives packets and all the packets go to the same destination. When in the “idle” state, it does not receive any packets. The length of the busy and idle periods follows geometric distribution. The network performance is measured by the *blocking probability* which is defined as the ratio of the number of rejected packets over the number of arrived packets. The durations of the connections are one time slot and for each experiment the simulation program is run for 100,000 time slots. As a comparison, the results for an other type of wavelength conversion, the “circular symmetrical” wavelength conversion which is slightly stronger than the ordered interval wavelength conversion is also shown.

In Figure 4 we plot the packet loss probability of the interconnect as a function of conversion distance. We tested under two traffic loads, $\rho = 0.6$ where average busy period 15 time slots and average idle period 10 time slots, and $\rho = 0.8$ where average busy period 40 time slots and average idle period 10 time slots. We can see that the blocking probability decreases as the conversion distance increases. But when the conversion distance is larger than a certain value, the decease of blocking probability is marginal. In this case there is little benefit for further increasing the conversion degree, which is exactly the reason for using limited range wavelength converters other than full range wavelength converters.



(a)

(b)

Fig. 4. Packet loss probability of WDM switch under bursty traffic where the packets have no priority. (a) 8 × 8 interconnect with 8 wavelengths per fiber. (b) 16 × 16 interconnect with 16 wavelengths per fiber.

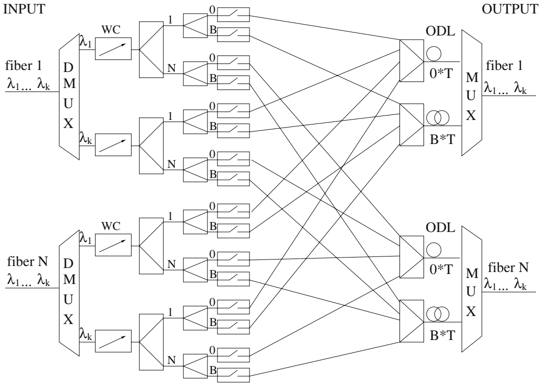


Fig. 5. A buffered wavelength convertible WDM optical switch.

5 Optimal Scheduling in Buffered WDM Switch

5.1 Network Model

A buffered WDM switch is shown in Fig.5. We can see that the only difference between Fig.5 and Fig.2 is the optical delay lines (ODL) placed in front of the output fibers. There are $B + 1$ optical delay lines, capable of delaying a packet for 0 to B time slots. The switching fabric is capable of connecting any input wavelength channel to any of the $B + 1$ ODLs for any output fiber. So if a packet that cannot be sent to the output fiber directly, it can be sent to one of the delay lines. A packet sent to delay line b will come out of the delay line after b time slots. The outputs of these $B + 1$ delay lines are directly combined together and sent to the output fiber.

Note that all the signals come out of the delay lines at the same time slot should be on different wavelengths. As a result, not all wavelength channels on the $B + 1$ ODLs are available to the new coming packets. Some of them, if assigned to the new packets, might cause collision later. Given the buffer occupancy state, we can find the set of available wavelength channels for newly arrived packets by scanning through the ODLs in linear time. To be specific, wavelength channel λ_i on ODL b denoted by λ_i^b is not available if we find that there is a packet that will come out of an ODL on wavelength λ_i after b time slots, for example, a packet on λ_i directed to ODL $b + 1$ at the previous time slot.

After getting the available wavelength channels, if only to maximize network throughput, we can simply run the First Available Algorithm described in the previous section. However, here we have another concern, since we also want to minimize the total delay, or, to send as many packets to shorter delays as possible. We can check the ODLs one by one, shorter ODLs first. By doing so we guarantee that the wavelength channels on shorter ODLs are given higher priorities. When checking ODL b, we should use as many wavelength channels on this ODL as possible, while making sure that all the wavelength channels checked previously on shorter delay lines that were assigned to some packets are still used, though not necessarily being assigned to the same packets.

To do so we can use the Scan and Swap Algorithm shown in Table 2. In the algorithm, the wavelength channels that were checked and assigned to packets prior to ODL b are

Table 2. Scan and Swap Algorithm

```
Set all packets as unmarked.
$\Pi \leftarrow \emptyset$.
for i=1 to n
    Find the first adjacent packet
    of $a_i$ that has not been marked.
    if there is such a packet
        $\Pi \leftarrow \Pi \cup \{a_i\}$
        Mark this packet.
    else
        if $a_i$ is a compulsory channel
            $\Pi \leftarrow \Pi \cup \{a_i\}$
            Let $a_s$ be the non-compulsory
            channel in $\Pi$ with the largest index
            $\Pi \leftarrow \Pi \setminus \{a_s\}$
        end if
    end if
end for
```

called the *compulsory channels*, and the available channels on ODL b are called the *non-compulsory channels*. For simplicity we use the a_i, $i \in [1, n]$ to denote the wavelength channels where n is the total number of compulsory channels and non-compulsory channels. Lower wavelengths are scanned first. That is to say, for two channels denoted by a_i and a_j, if a_i is on a lower wavelength then $i < j$. For channels on the same wavelength, the compulsory channels are checked prior to the non-compulsory channel. Compulsory channels on the same wavelength are checked in a arbitrary order. If the wavelength of a packet can be converted to a_i, we say this packet is *adjacent* to a_i. The *first* adjacent packet is the one on the lowest wavelength.

The algorithm outputs set Π initially set to be empty. Also, at the beginning all the packets is set to be unmarked. Then the algorithm starts scanning the channels from first to the last. When scanning to a_i, it checks if there is an unmarked packet adjacent to it a_i. If yes, it adds a_i to Π and mark the first such packet. Else it proceeds to the next channel if a_i is non-compulsory. Otherwise a_i is a compulsory channel, it adds a_i to Π and swap out a non-compulsory channel in Π with the largest index. When the algorithm terminates, Π will store the channels that can be assigned to the incoming packets. All the compulsory channels will be in Π, and the number of non-compulsory channels in Π is maximum.

This algorithm needs to be executed $B + 1$ times. The output of the b_{th} execution will be the compulsory channels of the $(b+1)_{th}$ execution. The output of the $(B+1)_th$ execution stores the wavelength channels that can be assigned to the incoming packets. Then an assignment can be found by running the First Available Algorithm on these channels and the incoming packets.

5.2 Complexity Analysis

We now show that the running time of the Scan and Swap Algorithm is $O(k)$ where k is the number of wavelengths.

The input to this algorithm are the set of compulsory channels, the set of non-compulsory channels and the set of packets. We can use a $1 \times k$ vector to represent each of these set, with element i in the vector being the number of channels or packets on wavelength λ_i. We will refer to them as the "compulsory vector", the "non-compulsory vector" and the "packet vector" and denote them as C, N and R, respectively.

When running the Scan and Swap Algorithm, the channels will be added to set Π if they can be covered along with all the vertices previously in Π. Since algorithm makes sure that all the compulsory channels are in Π, only the status of the non-compulsory channels needs to be recorded. For this, a stack can be used. When a non-compulsory channel is added to Π, its wavelength index will be pushed into this stack. In some later steps we may decide to swap out some of the non-compulsory channels. By the algorithm, they will be the ones that were most recently added to Π, i.e., will be on the top of the stack. Therefore to swap them out is simply to perform several pop operations. When the algorithm terminates the content in the stack will be the desired output.

In the algorithm the packets that were chosen to match to some wavelength channels need to be "marked". We can use a $1 \times k$ vector called the "marked vector", denoted as D, to represent the set of marked packets, with each element being the number of marked packets on the corresponding wavelength.

We can also use pointer p which is the wavelength index of a packet immediately following the packet that was just marked. That is to say, if the Scan and Swap Algorithm just marked a packet on λ_j, then $p = j$ if there are still unmarked packets on λ_j, otherwise $p = j + 1$. Initially $p = q$ where q is the smallest wavelength with $r_q > 0$.

A more detailed Scan and Swap Algorithm, with implementation issues considered, is shown in Table 3. The algorithm scans the wavelengths from λ_1 to λ_k. When scan to λ_i, first it checks whether the packet pointed by p is in the conversion range of λ_i by comparing p with $begin(i)$ and $end(i)$. If $p > end(i)$, all the packets convertible to λ_i must have all been marked or be used by the wavelength channels on lower wavelengths, since we always try to mark the first available packets for any channel. Therefore, if there are compulsory channels on λ_i or $c_i > 0$, they cannot mark any new packet. By the algorithm, we should swap out exactly c_i non-compulsory channels, which is to perform c_i *pop* operations on the stack. If $p < begin(i)$, the packet is also out of the conversion range of λ_i, however, in this case all the packets convertible to λ_i are not marked, and we set $p = begin(i)$.

Now the algorithm will try to find the first c_i packets that are in the conversion set of λ_i, which is done by the *while* loop. The loop exits if enough packets are found, or the last wavelength adjacent to λ_i has been reached.

In the second case, not enough packets were found, and we should *pop* the stack several times accordingly. Then we should update pointer p, by moving it to the first wavelength that has some packets.

In the first case, all the compulsory channels can find some packets to mark, and we go on to check the non-compulsory channel on λ_i, if there is such a channel ($n_i == 1$). If the packet pointed by p is within the conversion range of λ_i ($p \leq end(i)$) and there

Table 3. Scan and Swap Algorithm in Implementation

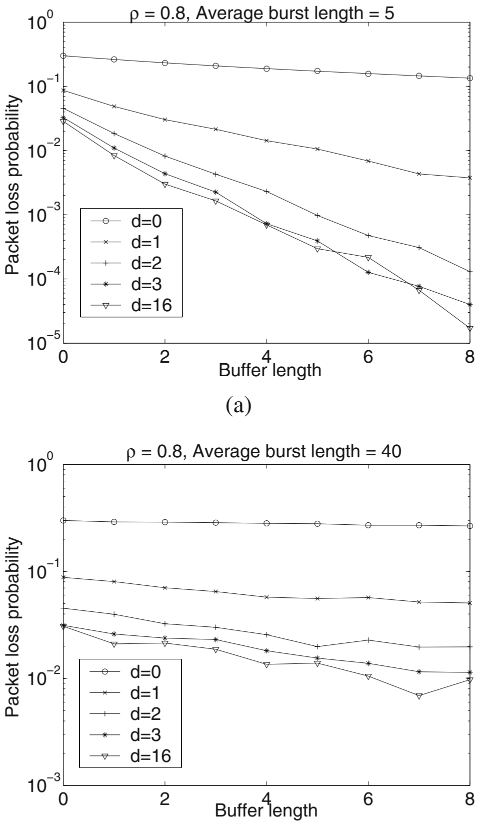
```
$p \leftarrow q$ where $q$ is the smallest wavelength with $r_q > 0$;
Set $D$ to be all zero vector;
for i=1 to k
    if $end(i) < p$
        pop stack $c_i$ times
    else
        if $begin(i) > p$
            $p \leftarrow begin(i)$;
        end if

        $\Delta \leftarrow r_p - d_p$ ;
        while $\Delta < c_i$
            $d_p \leftarrow r_p$ (mark all the packets on $\lambda_p$);
            $p \leftarrow p + 1$;
            if $p > end(i)$
                break;
            end if
            $\Delta \leftarrow \Delta + r_p$;
        end while

        if $p == end(i) + 1$
            pop stack $c_i - \Delta$ times
            set $p$ to the first $q$ where $p \leq q$ and $r_q > d_q$
        else
            $d_p \leftarrow r_p - \Delta + c_i$ (mark the needed packets on $\lambda_p$);
            set $p$ to the first $q$ where $p \leq q$ and $r_q > d_q$
            if $n_i == 1$ and $p \leq end(i)$ and $r_p - d_p > 0$
                push $i$ into the stack
                $d_p \leftarrow d_p + 1$ (mark a packet on $\lambda_p$)
                set $p$ to the first $q$ where $p \leq q$ and $r_q > d_q$
            end if
        end if
    end if
end for
```

is still an unmarked packet $d_p < r_p$, this non-compulsory channel can be added in and we *push* i into the stack. Then we update the mark vector and p accordingly.

Now we analyze the complexity of the Scan and Swap algorithm, when implemented in this way. First, the *push* and *pop* operations takes $O(k)$ time, since there are up to k non-compulsory channels, and a channel can be pushed in at most once because a channel that was *popped* out will never be *pushed* in again. For other operations within the *for* loop, except for the *while* loop, also need no more than $O(k)$ time. One execution of the *while* loop takes constant time. And, because each execution moves the pointer p down by one, the *while* loop is executed at most k times. Combining these we conclude that the running time of the Scan and Swap algorithm is $O(k)$.



(a)

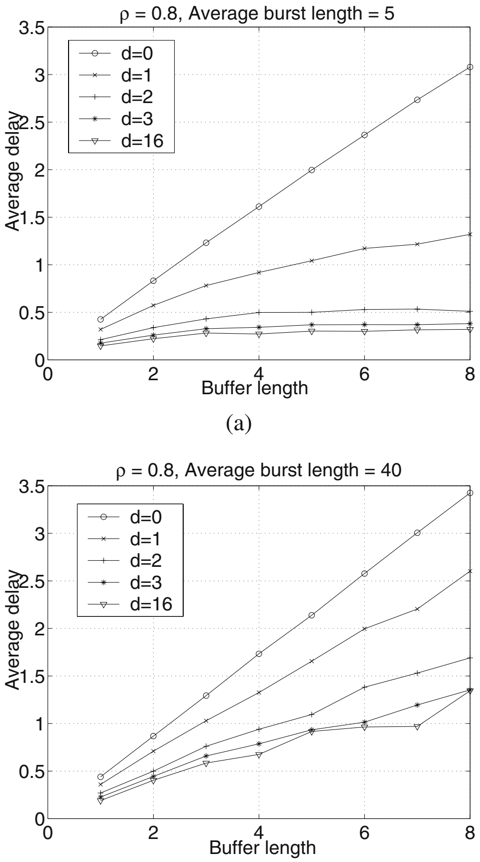
(b)

Fig. 6. Packet loss probability of the WDM switching network under bursty traffic. The load is $\rho = 0.8$. (a) Average burst length is 5 time slots. (b) Average burst length is 40 time slots.

In our applications the Scan and Swap Algorithm needs to run $B + 1$ times. Thus, when using this algorithm for optimal scheduling, we need $O(kB)$ time, where B is the length of the longest delay line and k is the number of wavelengths per fiber,

5.3 Simulation Results

We implemented Scan and Swap Algorithm in software and conducted simulations. The network in simulations has 16 input fibers and 16 output fibers with 16 wavelengths on each fiber. The arrivals of connection requests at input channels are bursty: an input channel alternates between two states, the “busy” state and the “idle” state. When it is in the “busy” state it continuously receives packets and all the packets go to the same destination; otherwise the input channel is in “idle” state and does not receive any packets. The length of the busy and idle periods follows geometric distribution. The network performance is measured by the *packet loss probability* which is defined as the



(a)

(b)

Fig. 7. Average delay of the WDM switching network under bursty traffic. The load is $\rho = 0.8$. (a) Average burst length is 5 time slots. (b) Average burst length is 40 time slots.

ratio of the total number of successfully transmitted packets over the total number of arrived packets. The durations of the packets are all one time slot and for each experiment the simulation program was run for 100,000 time slots.

In Fig. 6 we show the packet loss probability of the network as a function of the number of fiber delay lines. In Fig. 6(a) the average burst length is 5 time slots and the average idle period is 1.25 time slots. In Fig. 6(b) the average burst length is 40 time slots and the average idle period is 10 time slots. In both cases the traffic load is $\rho = 0.8$. As expected, packet loss probability decreases as the number of delay lines increases. For example, in Fig. 6(a), when the conversion distance $d = 2$, when $B = 0$ (no buffer), the packet loss probability is about $10^{-1.3}$. However, when $B = 4$, it is reduced to about 10^{-3}. As the traffic becomes more bursty, i.e., as the average burst length increases, the packet loss probability decreases much more slowly with the buffer depth, as can be observed in Fig. 6(b) where the curves are almost flat. This is because when the burst is too long it will always exceed the buffer capacity.

We can also notice that with the same buffer length, a larger conversion distance always results in a smaller packet loss probability. Also, when the burst is too long, increasing buffer length does not yield too much benefit, but increasing conversion distance always does. For example, in Fig. 6(b), when $d = 1$, increasing buffer length does not decrease much of the packet loss probability, but when we increase d to 2, the packet loss probability almost drops by $10^{-0.4}$. This suggests that wavelength conversion ability is more important than buffering in a WDM switching network. However, we can observe that only a relatively small conversion distance is needed to achieve good performance. As can be seen in Fig. 6, the packet loss probability for $d = 3$ is already very close to that for $d = 16$ (full range conversion). This is exactly the reason to use limited range wavelength converters instead of full range wavelength converters.

In Fig. 7 we show the average delay of a packet as a function of the number of fiber delay lines. The traffic is the same as in Fig. 6. We can see that as the buffer length increases the average packet delay also increases, since fewer packets are dropped and thus more are directed to a buffer before being actually transmitted. For the same buffer size, a larger conversion distance results in a shorter average delay. As in Fig. 7(a), when $B = 4$, the average delay for $d = 1$ is about 0.9 time slots and the average delay for $d = 3$ is only about 0.3 time slots.

6 Conclusions

In this paper we have presented optimal scheduling algorithms to resolve output contentions in time slotted WDM optical interconnects with limited range wavelength conversion ability. We gave the First Available Algorithm that runs in $O(k)$ time for finding an optimal scheduling in unbuffered interconnects where k is the number of wavelengths per fiber. We also gave the Scan and Swap Algorithm that runs in $O(Bk)$ time for finding an optimal scheduling in buffered interconnects where B is the buffer depth.

References

1. B. Mukherjee, “WDM optical communication networks: progress and challenges,” *IEEE Journal on Selected Areas in Communications*, vol. 18, no. 10, pp. 1810-1824, Oct. 2000.
2. D. K. Hunter, M. C. Chia and I. Andonovic “Buffering in optical packet switches,” *Journal of Lightwave Technology*, vol. 16 no. 12, pp. 2081-2094, 1998.
3. M. Kovacevic and A. Acampora, “Benefits of wavelength translation in all-optical clear-channel networks,” *IEEE Journal on Selected Areas in Communications*, vol. 14, no. 5, pp. 868 -880, June 1996.
4. S.L. Danielsen, C. Joergensen, B. Mikkelsen and K.E. Stubkjaer, “Analysis of a WDM packet switch with improved performance under bursty traffic conditions due to tuneable wavelength converters,” *Journal of Lightwave Technology*, vol. 16, no. 5, pp. 729-735, May 1998.
5. N. McKeown, “The iSLIP scheduling algorithm input-queued switch,” *IEEE/ACM Trans. Networking*, vol. 7, pp. 188-201, Apr. 1999.
6. W.J. Goralski, *Optical Networking and WDM*, 1st Edition, McGraw-Hill, 2001.

7. R. Ramaswami and K. N. Sivarajan, *Optical Networks: A Practical Perspective*, 1st Edition, Academic Press, 2001.
8. T. Tripathi and K. N. Sivarajan, “Computing approximate blocking probabilities in wavelength routed all-optical networks with limited-range wavelength conversion,” *IEEE Journal on Selected Areas in Communications*, vol. 18, pp. 2123–2129, Oct. 2000.
9. L. Xu, H.G. Perros and G. Rouskas, “Techniques for optical packet switching and optical burst switching,” *IEEE communications Magazine*, pp. 136 - 142, Jan. 2001.
10. R. Ramaswami and G. Sasaki, “Multiwavelength optical networks with limited wavelength conversion,” *IEEE/ACM Trans. Networking*, vol. 6, pp. 744–754, Dec. 1998.
11. Y. Yang and J. Wang, “WDM optical interconnect architecture under two connection models,” *Proc. of IEEE Hot Interconnects 10*, pp. 146-151, Palo Alto, CA, August 2002.
12. Y. Yang, J. Wang and C. Qiao “Nonblocking WDM multicast switching networks,” *IEEE Trans. Parallel and Distributed Systems*, vol. 11, no. 12, pp. 1274-1287, 2000.
13. Z. Zhang and Y. Yang, “Distributed scheduling algorithms for wavelength convertible WDM optical interconnects,” *Proc. of the 17th IEEE International Parallel and Distributed Processing Symposium,* Nice, France, April, 2003.
14. E.L. Lawler, “Combinatorial Optimization:Networks and Matroids,” *Holt, Rinehart and Winston*, 1976.
15. F. Glover “Maximum matching in convex bipartite graph,” *Naval Res. Logist. Quart.*,14, pp. 313-316, 1967.
16. W. Lipski Jr and F.P. Preparata “Algorithms for maximum matchings in bipartite graphs,” *Naval Res. Logist. Quart.*,14, pp. 313-316, 1981.
17. G. Shen, et. al, “Performance study on a WDM packet switch with limited-range wavelength converters,” *IEEE Communications Letters* , vol. 5, no. 10, pp. 432-434, Oct. 2001.

Comparative Evaluation of Adaptive and Deterministic Routing in the OTIS-Hypercube

Hashem Hashemi Najaf-abadi[1] and Hamid Sarbazi-Azad[1,2]

[1] School of Computer Science, IPM, Tehran, Iran.
{h_hashemi, azad}@ipm.ir
[2] Computer Eng. Department, Sharif Univ. of Tech., Tehran, Iran.
azad@sharif.edu

Abstract. The OTIS-hypercube is an interesting class of the optoelectronic OTIS architecture for interconnection networks. In the OTIS architecture, optical connections are used to connect distant processors while closer processors are connected electronically. In this paper, we propose an adaptive routing algorithm for the wormhole switched OTIS-hypercube. We then present an empirical performance evaluation of adaptive wormhole routing in these networks for different structural conditions and traffic loads. The effect of maximum wire length and router delay on performance measures, such as average message latency and bandwidth of the interconnection network, are also briefly brought into consideration and compared with those of equivalent hypercubes. In addition, the performance merits of adaptive wormhole routing in the OTIS-hypercube are compared with those of deterministic routing using extensive simulation experiments.

1 Introduction

The conveyance of data between processing elements, in an interconnection network, is usually made through electrical conductance. But where communication distance exceeds a few millimeters, optical interconnect provides speed and power advantages over electronic interconnect [2, 3]. Therefore, in the design of very large multiprocessor systems, to interconnect physically close processors using electronic interconnect and to use optical interconnect for pairs of processors that are distant, seems a reasonable option. Marsden et al. [4], Hendrick et al. [5] and Zane et al. [6] have proposed such an architecture named the OTIS (Optical Transpose Interconnect System) in which the processors are partitioned into groups. Within each group electronic interconnect is used to connect the processors, and optical interconnect is used to bring about connection between processors in different groups.

The significance of an architecture depends on whether that architecture can be used to effectively solve problems that are of interest. The development of algorithms for OTIS-based computers has been the focus of much attention. Algorithms for OTIS-hypercubes have been developed by Sahni and Wang [10]. Algorithms for OTIS-mesh computers (in this type of OTIS each group is a mesh instead of a hypercube) have been studied more extensively by Zane *et al* [6], Sahni and Wang [11, 12, 13, 14], Rajasekeran and Sahni [15] and Osterloh [16]. Sahni and Wang [11]

P.-C. Yew and J. Xue (Eds.): ACSAC 2004, LNCS 3189, pp. 349–362, 2004.

have also developed a routing algorithm for the OTIS-mesh network. However, no work has, to our best knowledge, investigated the appropriateness of these systems for general purpose applications using realistic implementation assumptions, i.e. these studies have considered topological and algorithmic issues in OTIS computers and no study has been conducted to evaluate the performance of these systems in sight of parameters such as bandwidth and message latency.

The layout of the paper is as follows. OTIS parallel computers and specifically the OTIS-hypercube network are first described in detail. A deadlock-free adaptive routing algorithm is then suggested for this network. Simulation results of different cases of the network with different traffic loads are compared and a number of observations, based on these results, are made. Finally, an overview of the experimental results is provided and some concluding remarks are given.

2 Preliminaries

The reader is referred to [8] for an in-depth account of basic concepts such as topology, routing algorithms, flow control, wormhole switching and virtual channels. In this section, more specific concepts are described.

2.1 The OTIS-Hypercube Interconnection Network

In order to interconnect physically close processors using electronic interconnect and distant processors with optical interconnect, various combinations of interconnection networks have been proposed. In OTIS computers, optical interconnects are realized via a free space optical interconnect system proposed by Marsden et al. [4]. In this system, processors are partitioned into groups of the same size. Krishnamoorthy et al. [17] have shown that when the number of groups equals the number of processors within a group, the bandwidth and power efficiency are maximized, and system area and volume are minimized.

In the OTIS-hypercube parallel computer, there are 2^{2N} processors organized as 2^N groups of 2^N nodes each. The processors in each group form an N dimensional hypercube that employs electrical interconnect. The inter-group interconnections are realized by optics. In the OTIS interconnect system, processor (i, j), i.e. processor j of group i, is connected via optics to processor (j, i). A partial 3-dimensional OTIS-hypercube is illustrated in Fig. 1. In this figure, the optical interconnections corresponding to group 0 are shown by dashed lines. Electronic interconnections in each group are shown by solid lines. The address of each group is placed in parentheses above the group. The address of each node in group (001) is displayed near the node, and the nodes in other groups are assigned addresses in the same order.

2.2 Node Structure in the OTIS Architecture

A node, in the n-dimensional OTIS-hypercube, or OTIS-H_n for short, consists of a processing element (PE) and a switching element (SE), as illustrated in Fig. 2. The PE

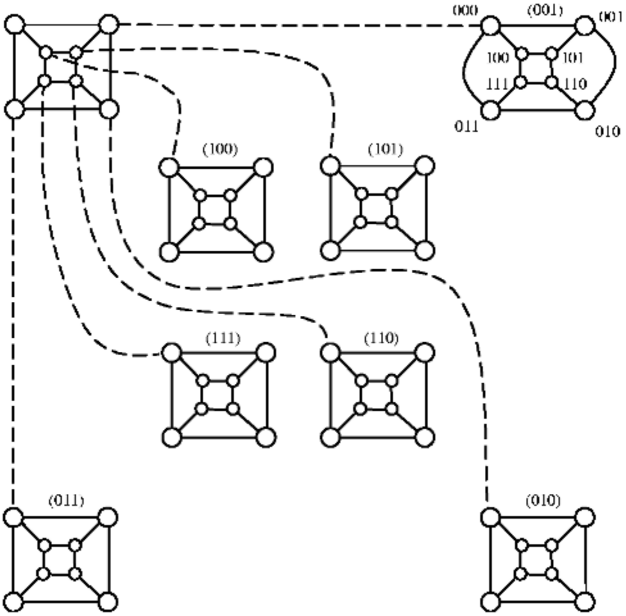


Fig. 1. A 3-dimensional OTIS-hypercube with the optical connections exiting one of the sub-graphs (numbers inside parenthesis are sub-graph addresses)

contains a processor and some local memory. A node is connected, through its SE, to its intra-group neighboring nodes using *n* input and *n* output electronic channels. Two electronic channels are used by the PE to inject/eject messages to/from the network. Messages generated by the PE are transferred to the router through the injection channel. At the destination node, messages are transferred to the local PE through the ejection channel. The optical channel is used to connect a node to its transpose node in another group for inter-group communication. The router contains flit buffers for each incoming channel. A number of flit buffers are associated with each physical input channel. The flit buffers associated with each channel may be organized into several lanes (or virtual channels), and the buffers in each virtual channel can be allocated independently of the buffers in any other virtual channel [8]. The concept of virtual channels has been first introduced in the context of the design of deadlock free routing algorithms, where the physical bandwidth of each channel is multiplexed between a number of messages [7, 8]. However, virtual channels can also reduce network contention. This is while it has been shown that virtual channels are expensive, increasing node delay considerably [1]. So, the number of virtual channels per physical channel should be reasonable. The input and output virtual channels are connected by a crossbar switch that can simultaneously connect multiple input channels to multiple output channels given that there is no contention over the output channels.

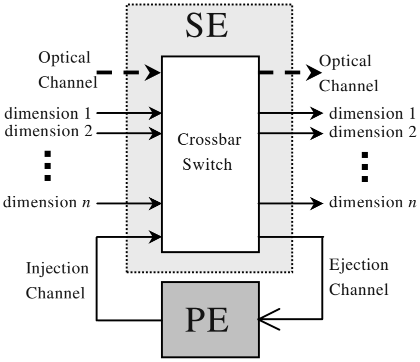


Fig. 2. The node structure in the OTIS-hypercube

3 Deadlock-Free Wormhole Routing in the OTIS-Hypercube

3.1 Deadlock-Free Adaptive Routing

With fully adaptive routing, the header of a message can be routed through any dimension that takes the message closer to its destination. But the actual dimension that the message is routed through depends on which dimension has a free virtual channel at the time. This is contrary to deterministic routing in which dimensions are traversed strictly in a predetermined order.

In the case of OTIS interconnection networks, two basic adaptive routing algorithms can be suggested. In the first algorithm, a message is routed adaptively in the local sub-graph (group) in which it starts until it reaches the node that has the same node address as the destination node. From that node, the optical channel is taken into another sub-graph. In this sub-graph, the message is routed (once again adaptively) until it reaches a node that has the same node address as the sub-graph address of the destination node. Once there, the message takes its final optical hop to the destination node. In the second algorithm, a message is first adaptively routed to a node that has a node address equal to the sub-graph address of the destination. Once there, the optical channel takes the message to the sub-graph of the destination node. The message is then adaptively routed to the destination node within this sub-graph. It is obvious that, although labeled adaptive, both algorithms make use of the optical channels deterministically. The reason for this is that utilization of the optical channels in any other manner will cause the routing algorithm to become non-minimal.

Of the former algorithms, the one that takes a shorter path depends on the full address (consisting of the sub-graph and node addresses) of the source and destination nodes. When considering the OTIS-hypercube, this can be determined effortlessly. If the number of differing bits of the full address of the source and destination nodes is less than the number of differing bits of the full address of the source node and the transpose of the address of the destination node, then the first routing algorithm will result in a shorter path. Otherwise, the second algorithm will. However, it should be obvious that in the first routing algorithm, once the first optical channel has been taken, the rest of the routing is identical to that of the second algorithm. Therefore, if in each node, a message is routed according to the algorithm that takes a shorter path to the destination of the message (without considering the source node), a more global optimal routing algorithm is obtained.

In order for a routing algorithm to be deadlock-free, cyclic buffer dependencies between messages and the virtual channels they allocate, must not occur. Duato [18] has suggested a virtual channel allocation scheme for fully-adaptive deadlock-free routing in the hypercube. In that scheme, all virtual channels except one are used for adaptive routing, and the leftover virtual channel is used only by messages that are traversing the lowest dimension they must traverse (just as they would have, if routed deterministically). But in an OTIS-hypercube, the optic channels between the sub-graphs (hypercubes) may also cause cyclic dependencies if no further restriction is enforced on the virtual channels a message may traverse. To prevent the occurrence of such cyclic dependencies, the virtual channels of each electronic physical channel must be divided into two sets, i.e. each sub-graph must be split into two virtual networks. A message, after being injected into the network, traverses the source sub-graph through its first virtual network. But once an optical channel has been traversed and the message has entered another sub-graph, that sub-graph is traversed through its second virtual network.

According to the minimal routing algorithm described above, each message traverses either one or two optical channels. Cyclic dependencies between messages that traverse one optical channel can obviously not occur. But, for messages that traverse two optical channels, that would not have been the case, had there been no restriction on the usage of the virtual optical channels. Since a message that has traversed its second optical channel has definitely entered its destination node, it can not be part of a cyclic buffer dependency. Therefore, reserving one of the virtual channels of each optical channel, specifically for such messages, eliminates the possibility of the occurrence of cyclic buffer dependencies.

In each virtual network of each sub-graph, virtual channel allocation is performed according to Duato's scheme. Thus, since the minimum number of virtual channels needed to implement Duato's scheme is equal to two (with one virtual channel, routing becomes fully deterministic) and there are two virtual networks per sub-graph, the minimum number of virtual channels needed for adaptive routing in the OTIS-hypercube is equal to four.

Fig. 3 displays the pseudo code of the minimal adaptive routing algorithm. In this code, all messages are considered to be inserted into the 0th virtual network (*InNet*=0). The ‖ Offset ‖ operator returns the number of one's in the binary representation of Offset, and the SelectOne( ) function adaptively selects a dimension, that has a free virtual channel, corresponding to a one in the binary representation of the input parameter. The ChannelsOfNet() function is considered to return the set of virtual channels of the virtual network determined by the parameter passed to it. The SelectVirtualChannel() function selects one of the free virtual channels passed to it. The names of the other functions are descriptive of their operation.

```
Algorithm: Adaptive deadlock-free routing in an n-D OTIS-hypercube
Inputs: Coordinates of current node ($Current_{Sub}$, $Current_{Node}$) and
        destination node ($Dest_{sub}$, $Dest_{node}$) and input virtual network InNet.
Output: Selected output physical and virtual channel, [$p_c$, $v_c$].
Procedure:
$Offset_{sub-sub} = Current_{Sub} \oplus Dest_{Sub}$
$Offset_{Node-Node}=Current_{Node} \oplus Dest_{Node}$
$Offset_{Node-Sub}=Current_{Node} \oplus Dest_{Sub}$
$Offset_{Sub-Node}=Current_{Sub} \oplus Dest_{Node}$

If InFromOptic then
  OutNet := 1;
Else
  OutNet := InNet;
Endif

If $\|Offset_{Sub-Sub}\| + \|Offset_{Node-Node}\| \leq \|Offset_{Sub-Node}\| + \|Offset_{Node-Sub}\|$ then
  If $Offset_{Node-Node} \neq 0$ then
    $P_c$ := SelectOne ($Offset_{Node-Node}$);
    If $P_c$ = LeastSignificantOne ($Offset_{Node-Node}$) then
      $V_c$ := SelectVirtualChannel ( ChannelsOfNet ( OutNet ) );
    Else
      $V_c$ := SelectVirtualChannel ( ChannelsOfNet ( OutNet ) – {0} );
    Endif
  Else
    If $Offset_{Sub-Sub} \neq 0$ then
      If OutNet = 1 then
        $P_c$ := Optical; $V_c$ := SelectVirtualChannel (Any);
      Else
        $P_c$:= Optical;
        $V_c$ := SelectVirtualChannel( Any – {$Channel_{reserved}$});
      Endif
    Else
      $P_c$ := Internal; $V_c$ := SelectVirtualChannel(Any);
    Endif
  Endif
Else
  If $Offset_{Node-Sub} \neq 0$ then
  $P_c$ := SelectOne($Offset_{Node-Sub}$);
    If $P_c$ = LeastSignificantOne($Offset_{Node-Sub}$) then
      $V_c$ := SelectVirtualChannel(ChannelsOfNet(OutNet));
    Else
      $V_c$ := SelectVirtualChannel(ChannelsOfNet(OutNet) – {0});
    Endif
  Else
    If OutNet = 1 then
      $P_c$ := Optical; $V_c$ := SelectVirtualChannel(Any);
    Else
      $P_c$ := Optical;
      $V_c$ := SelectVirtualChannel ( Any -{$Channel_{reserved}$});
    Endif
  Endif
Endif
```

Fig. 3. Adaptive deadlock-free wormhole routing algorithm for the OTIS-hypercube

3.2 Deadlock-Free Deterministic Routing

With deterministic routing, the routing within each sub-graph is performed according to a deterministic algorithm such as e-cube routing. However, for deadlock-free deterministic routing, the minimum number of virtual channels per physical channel in a hypercube is equal to one. Thus the minimum number of necessary virtual channels for deadlock-free deterministic routing in an OTIS-hypercube is equal to two.

3.3 Deadlock-Free Partially Adaptive Routing

A class of deadlock-free routing algorithms for the hypercube (that are not based on adding virtual channels to the structure of the network) are the partially adaptive routing algorithms, such as west-first and p-cube routing, presented by Glass and Ni [19]. A partially adaptive routing algorithm in the OTIS-hypercube is such that routing within each sub-graph is performed according to one of these partially-adaptive routing algorithms. One virtual channel is sufficient for the implementation of these algorithms in a hypercube. Thus, two virtual channels are sufficient to implement such partially adaptive routing algorithms in the OTIS-hypercube. But, as pointed out in [19], these partially-adaptive routing algorithms do not perform better than deterministic routing for random uniform traffic. Therefore, in what follows, we consider only the performance merits of adaptive and deterministic routing.

4 Empirical Performance Evaluation

To evaluate the functionality of the OTIS-hypercube network under different conditions, a discrete-event simulator has been developed that mimics the behavior of the described adaptive routing algorithm at the flit level in the OTIS-hypercube network. The simulator has been coded in C++ and consists of a number of different classes which are, at the highest hierarchical level, used to define the *network* class. Specifically, in the constructor of the *network* class, objects of the *node* and *channel* classes are defined. Each *node* has a number of pointers to its input and output channels, and each *channel* has a pointer to the *node* it is an input channel to. The way these pointers are initialized determines the topology of the network to be simulated. Once the *network* has been constructed, messages are injected into the network by *injection* events.

Four different types of *event* classes have been defined, namely, *injection*, *header-routing*, *handshaking* and *channel-switching* events. An event is specified to occur at a particular time and location. At each instance of time, all the events that must be executed at that time are completed. Only then is the time counter incremented and the events of the next time instance executed. This is while the execution of an event may generate another event to be executed at a future time. For instance, the execution of an injection event generates a header-routing event. That event, when executed, causes a channel-switching and a handshaking event.

When a flit is transferred from one buffer to the next, a counter at the corresponding virtual channel is decremented and a handshaking event, to be executed at the next time unit, is produced. This event notifies the preceding buffer allocated by the message that there is an empty space ahead into which it can transfer a new flit. A header-routing event, to be executed at a time determined by the channel cycle time of the network, is also produced. Once routed to a specific channel, a message waits until that channel is switched by a channel-switching event. If there is a free virtual channel available at the time of switching, it is allocated to that message and a counter corresponding to the virtual channel is initialized to the length of the message. The virtual channel on a physical channel is switched in a round-robin manner and every time a virtual channel is switched onto the physical channel, the corresponding counter is decremented if the buffer of that channel has a new flit and that of the next allocated virtual channel is empty. The execution of all these events put together, results in the simulation of the functionality of the entire interconnection network.

In each simulation experiment, a minimum of 120,000 messages have been delivered and the average message latency calculated. Statistics gathering was inhibited for the first 10,000 messages to avoid distortions due to startup transience. The mean message latency is defined as the average amount of time from the generation of a message until the last data flit of that message is consumed at the local PE at the destination node. The network cycle time is defined as the transmission time of a single flit from one router to the next, through an electric channel. The transmission time of a flit, through an optical channel is however a fraction of the network cycle time. In what follows, the ratio of optical channel transmission time to the network cycle time is referred to as the *channel cycle ratio*. Messages are generated at each node according to a Poisson process with a mean inter-arrival rate of λ_g messages per cycle. All messages have a fixed length of *M* flits. The destination node of each message has been determined through a uniform random number generator to simulate a uniform traffic pattern.

Numerous experiments have been performed for several combinations of network size, message length and number of virtual channels. The results of which are studied in the following subsections.

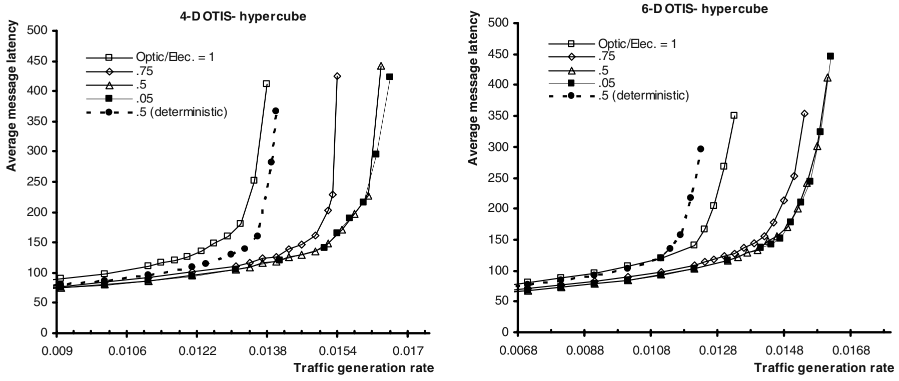


Fig. 4. Average message latency in OTIS-hypercubes with 4 virtual channels per physical channel, message length of 32 flits, and different channel cycle ratios; (a) 4-dimensional OTIS-hypercube, and (b) 6-dimensional OTIS-hypercube

4.1 The Effect of Channel Cycle Ratio

Fig. 4 depicts message latency results for the 4-dimensional and 6-dimensional OTIS-hypercubes for different cases of the *channel cycle ratio* with the number of virtual channels per physical channel equal to 4. It is evident from these figures that decreasing the channel cycle ratio results in an increase in the generation rate for which saturation occurs. But the effect gradually diminishes and from a point onwards, reducing the ratio any further has no effect on the saturation point.

In these figures, results obtained from deterministic routing when the effect of decreasing the channel cycle ratio is maximum, are also displayed to illustrate the fact that the effect of adaptivity is generally greater than the maximum effect of decreasing the channel cycle ratio.

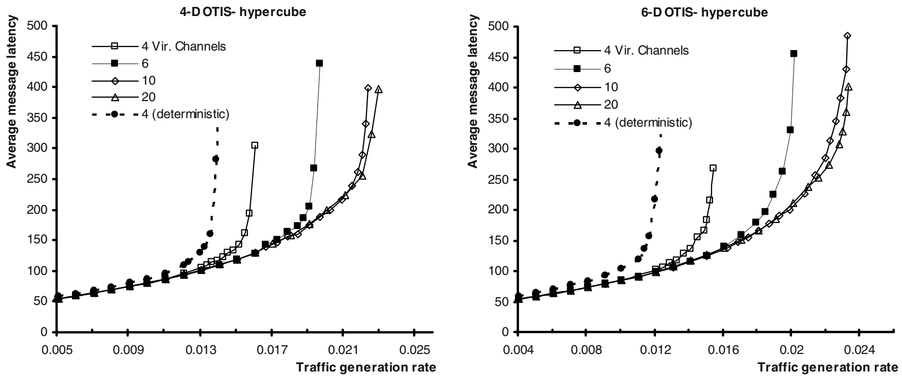
4.2 The Effect of the Number of Virtual Channels

Fig. 5 shows the average message latency of 4-dimensional and 6-dimensional OTIS-hypercubes, with a channel cycle ratio of 0.1, for different numbers of virtual channels per physical channel and a message length of 32 flits. It is observed that, increasing the number of virtual channels initially causes a considerable increase in the generation rate for which saturation occurs, but gradually looses its effect. Eventually, the saturation point reaches the bandwidth of the system. At this point, increasing the number of virtual channels, no longer has any effect on the saturation point.

It is evident from Fig. 5 that the generation rate of the saturation point becomes equal to the bandwidth of the corresponding network when the number of virtual channels per physical channel is equal to 10. With this number of virtual channels, the network saturates with a generation rate of 0.022 messages per node per cycle. It can therefore be concluded that the bandwidth of a 4-D OTIS-hypercube (with messages 32 flits long) is approximately equal to 0.022. In the same figure the message latency of deterministic routing in an OTIS-hypercube with a small number of virtual channels is also depicted. This shows that, although two virtual channels are sufficient to implement deterministic routing in the OTIS-hypercube, deterministic routing performs considerably worse than adaptive routing even with four virtual channels.

The average message latency of a number of different sized OTIS-hypercubes (with a large number of virtual channels and a *channel cycle ratio* of 0.1), are depicted in Fig. 6. From this figure, it is evident that the bandwidth of the OTIS-hypercube is almost independent of its size. Therefore, in regard to performance, the OTIS-hypercube can be considered to be a scalable architecture.

Another network that possesses a very high degree of performance scalability is the hypercube. This can also be observed in the results of Fig. 6 where the average message latency of 6-D and 8-D hypercubes are depicted. These results show that when not considering implementation constraints (considering the channel cycle time of the fully-electronic hypercube and OTIS-hypercube to be the same), the OTIS-hypercube possesses less bandwidth than a hypercube with the same number of nodes. This may be due to the smaller number of channels in an OTIS-hypercube compared to an equivalent hypercube (with the same number of nodes). But the interesting point is that *when the channel cycle ratio is reduced, the maximum bandwidth of the OTIS*



(a) **(b)**

Fig. 5. Average message latency of OTIS-hypercubes with a channel cycle ratio of 0.1 and a message length of 32 flits, for different numbers of virtual channels per physical channel; (a) 4-dimesnional OTIS-hypercube, and; (b) 6-dimesnional OTIS-hypercube

hypercube becomes almost equal to the bandwidth of an equivalent hypercube. This is while, in a similar comparison of deterministic routing in hypercube and OTIS-hypercube networks, the maximum bandwidth of the OTIS-hypercube is found to be considerably less than that of the equivalent hypercube.

4.3 The Effect of Implementation Constraints

When implementation constraints are brought into account, considerable degradation in the performance of the hypercube becomes apparent as a result of the lengthy transmission time of long wires. But in the OTIS-hypercube, long electronic interconnections do not exist. The maximum channel transmission time of the OTIS-hypercube is therefore a fraction of that of a same sized hypercube. In Fig. 6, the average message latency of the 3-D OTIS-hypercube (with a channel cycle ratio of 1.0) is once again compared with that of an equivalent hypercube. This time however, the network cycle time of the OTIS-hypercube has been scaled to different fractions of the cycle time of the hypercube. Considering the performance scalability of the hypercube and OTIS-hypercube, and the fact that similar results to that of Fig. 7 have been obtained for different message lengths, it can be concluded that for the bandwidth of an OTIS-hypercube (with channel cycle ratio equal to 1.0) to be comparable to that of an equivalent hypercube, it is sufficient that the network cycle time of the OTIS-hypercube be roughly half that of the hypercube.

The network cycle time of a network consists of two important attributes, namely, intra-node and inter-node latency. The inter-node delay time depends primarily on topology and packaging. Specifically, the inter-node delay is proportional to the maximum wire length in the layout of the network. It is shown in [20] that the maximum wire length of the most compact layout, for the hypercube network, is equal to $N/3+o(N)$. Therefore, inter-node delay in the hypercube is linearly proportional to the number of dimensions of the network. In an OTIS-hypercube, the

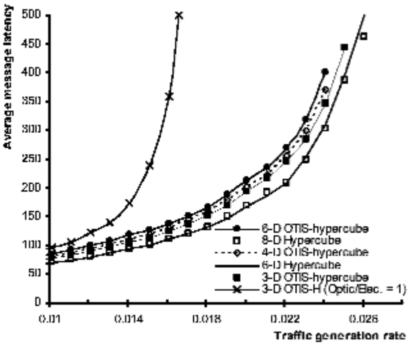


Fig. 6. The average message latency of equivalent hypercubes and OTIS-hypercubes with 30 virtual channels per physical channel (the default channel cycle ratio is 0.1).

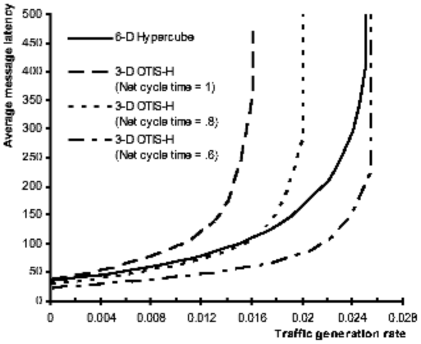


Fig. 7. The average message latency of a 6-dimensional hypercube and its equivalent 3-dimensional OTIS-hypercube (with 30 virtual channels) for different values of the network cycle

number of dimensions of each sub-graph is equal to half that of an equivalent hypercube. Thus, the maximum wire length (electronic) in an OTIS-hypercube is approximately half that of an equivalent hypercube. On the other hand, according to [1], the intra-node delay of a network (which is dominated by the crossbar and router delays) is logarithmically proportional to the number of input and output channels to a node in the network.

Therefore, in an OTIS-hypercube network, a channel cycle time equal to half that of an equivalent hypercube may easily be achievable when the inter-node delay is the dominant delay factor. Even then, decreasing the channel cycle ratio will result in further superior performance by the OTIS-hypercube.

4.4 Performance-Cost Analysis

When the performance to cost ratio of the OTIS-hypercube is compared to that of an equivalent hypercube, it is observed that, for low generation rates, the OTIS-hypercube is superior to the hypercube, when the channel cycle ratio is equal to 1.0 (assuming the use of electronic channels for transpose communication). This is shown in Fig. 8, where the inverse of average message latency is considered to be representative of performance, and the number of physical channels entering (or exiting) the nodes of a network is considered to be representative of cost. From these results, it can be concluded that compared to a hypercube, the OTIS-hypercube topologically performs better at a lower cost for low generation rates.

4.5 An Adaptive-Deterministic Comparison

As a rule, the bandwidth of a network is independent of the routing algorithm used in the network. In other words, with a large number of virtual channels, the generation

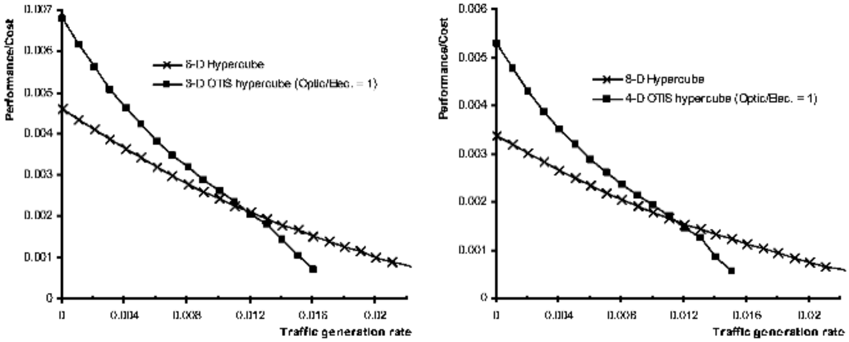


Fig. 8. Performance to cost ratio of adaptive routing in the hypercube compared to that of the OTIS-hypercube with a large number of virtual channels and the channel cycle ratio equal to 1.0

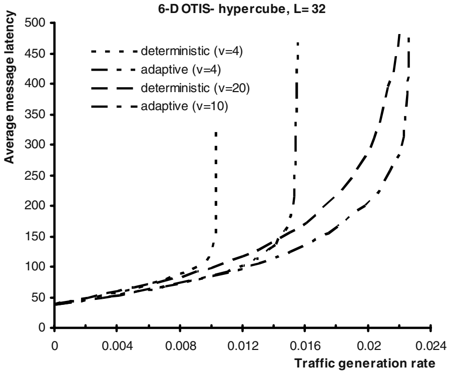


Fig. 9. OTIS-hypercube average message latency with different numbers of virtual channels per physical channel for adaptive and deterministic routing (the default channel cycle ratio is 0.1).

average message latency of deterministic and adaptive routing in the OTIS-hypercube are compared. The difference between adaptive and deterministic routing is more noticeable when there are a small number of virtual channels per physical channel. It is evident from the results of Fig. 9 that the network saturates at a higher generation rate with adaptive routing. But since the routing algorithm has no influence on the bandwidth of the network, a straightforward conclusion is that with adaptive routing fewer virtual channels are needed for the saturation point of the network to reach its maximum (the bandwidth of the network).

5 Conclusions

Unlike previous studies which have considered the topological and algorithmic issues in OTIS computers, in this study, we have investigated these systems in view of more realistic assumptions. An adaptive deadlock-free wormhole routing algorithm for the OTIS-hypercube is presented and the performance of this algorithm under uniform traffic is studied and compared to that of a deterministic routing algorithm.

Results reveal that decreasing the ratio of the optic to electronic channel transmission time is only of significance when the ratio is approximately less than half. In view of performance, the OTIS-hypercube is observed to be a scalable network. It has been shown that, for the bandwidth of an OTIS-hypercube to be comparable to that of a same sized hypercube, it is sufficient that the network cycle time of the OTIS-hypercube be half that of the hypercube. When implementation factors are accounted for, we find this condition to be easily attainable. It has also been shown that, even when the network cycle time of the OTIS-hypercube and the optical channel transmission time are equal to that of an equivalent fully-electronic hypercube, the performance-to-cost ratio of the OTIS-hypercube is higher than that of the hypercube for low generation rates. Finally, the bandwidth of the network is found to be independent of the routing algorithm used. Therefore, with adaptive routing fewer virtual channels are needed for the maximum saturation point to be attained.

Our next objective is to conduct an analysis of the OTIS-hypercube under different traffic patterns with different routing algorithms in order to evaluate the effect of adaptivity and minimality of routing, on performance.

References

1. A. A. Chien, "A cost and speed model for k-ary n-cube wormhole routers", *In Proceedings of Hot Interconnects'98*, August 1993.
2. M. Feldman, S. Esener, C. Guest and S. Lee, "comparison between electrical and free space optical interconnects based on power and speed considerations", *applied optics*, 27(9): 1742-1751, May 1988.
3. F. Kiamilev, P. Marchand, A. Krishnamoorthy, S. Esener, and S. Lee, "performance comparison between optoelectronic and VLSI multistage interconnection networks", *journal of lightwave technology*, 9(12): 1674-1692, Dec. 1991.
4. G. C. Marsden, P. J. Marchand, P. Harvey, and S. C. Esener, "Optical transpose interconnect system architectures", *Optical Letters*, 18(13): 1083-1085, July 1993.
5. W. Hendrick, O. Kibar, P. Marchand, C. Fan, D. V. Blerkom, F. McCormick, I. Cokgor, M. Hansen, and Esener, "modeling and optimization of the optical transpose interconnection system", *In optoelectronic technology Center*, Program Review, Cornel University, Sept. 1995.
6. F. Zane, P. Marchand, R. Paturi, and S. Esener, "Scalable network architectures using the optical transpose interconnection system (OTIS)", *In proceedings of the second International Conference on Massively Parallel Processing using Optical Interconnections* (MPPOI'96), pages 114-121, San Antonio, Texas, 1996.
7. W.J. Dally and C. Seitz, "Deadlock-free message routing in multiprocessor interconnection networks", *IEEE Trans. Computers,* 36 (5) (1987), 547-553.
8. W.J. Dally, "Virtual channel flow control", *IEEE Trans. Parallel and Distributed Systems*, 3 (2) (1992), 194-205.
9. J. Duato, "Why commercial multicomputers do not use adaptive routing", *IEEE Technical Committee on Computer Architecture Newsletter*, (1994), 20-22.
10. S. Sahni, C.-F. Wang, "BPC permutations on the OTIS-hypercube optoelectronic computer", *Informatica*, 22: 263-269, 1998.
11. S. Sahni and C.-F. Wang, "BPC permutations on the OTIS-mesh optoelectronic computer", In proceedings of the fourth international conference on massively parallel processing using *optical interconnections* (MPPOI'97), pages 130-135, 1997.

12. C.-F. Wang and S. Sahni, “Matrix multiplication on the OTIS-mesh optoelectronic computer”, *In Proceedings of the sixth international conference on Massively Parallel Processing using Optical Interconnections* (MPPOI’99), pages 131-138, 1999.
13. C.–F. Wang and S. Sahni, “Image processing on the OTIS-mesh optoelectronic computer”, *IEEE transaction on parallel and distributed systems*, 11(2): 97-107, 2000.
14. C.–F. Wang and S. Sahni “Basic operations on the OTIS-mesh optoelectronic computer”, *IEEE transaction on parallel and distributed systems*, 9(12): 1226-1236, 1998.
15. S. Rajasekeran and S. Sahni “Randomized routing, selection and sorting on the OTIS-mesh”, *IEEE transaction on parallel and distributed systems*, 9(9): 833-840, 1998.
16. A. Osterloh, “Sorting on the OTIS-mesh”, *In Proceedings of the 14th International Parallel and Distributed Processing Symposium* (IPDPS’2000), pp. 269-274, 2000.
17. Krishnamoorthy, P. Marchand, F. Kiamilev, and S. Esener, “Grain–size considerations for optoelectronic multistage interconnection networks”, *Applied Optics*, 31(26): 5480-5507, Sept. 1992.
18. J. Duato, T. Pinkston, “A general theory for deadlock-free adaptive routing using a mixed set of resources”, IEEE Transaction on Parallel and Distributed Systems, Vol. 12, 2001, pp. 1219-1235.
19. L. Ni and C. Glass, “The Turn Model for Adaptive Routing”, In Proc. of the 19th International Symposium on Computer Architecture, IEEE Computer Society, pp. 278-287, May 1992.
20. Yeh, C.-H., E.A. Varvarigos, and B. Parhami, “Efficient VLSI layouts of hypercubic networks,” *Proc. Symp. Frontiers of Massively Parallel Computation*, Feb. 1999, pp. 98-105.

A Two-Level On-Chip Bus System Based on Multiplexers

Kyoung-Sun Jhang[1], Kang Yi[2], and Soo Yun Hwang[1]

[1] Dept. of Computer Eng., ChungNam National University
sun@cnu.ac.kr, charisma@ce.cnu.ac.kr
[2] School of Computer Sci. and Electronic Eng., Handong Global University
yk@handong.edu

Abstract. The SoC (System on a Chip) design paradigm becomes a promising way of system integration as the level of design complexity is getting higher. There may be many IP modules to be integrated on a single chip in the modern SoC design. On-chip buses are usually used to interconnect the modules on a chip. Many bus architectures have been proposed for the interconnection of modules on a chip. We propose a two-level on-chip bus system that provides inter-bus transactions with multiplexers rather than with tri-state buffers or MOS switches such as in the segmented bus approaches. Our bus system can maximize the system throughput with concurrent inter-bus transactions as well as intra-bus transactions while preserving the already developed on-chip bus protocols for IP reuse. We present the performance simulation results of our approach with several different configurations compared with the existing segmented bus structures in terms of the total number of bus transactions executed in a given time.

1 Introduction

Today's deep submicron fabrication technologies enable design engineers to place billions of transistors on a single chip. These high-integrated circuit technologies make it possible for designers to integrate a number of function blocks like processors, memories, interfaces, and custom logic on a single chip. As the number of IP blocks increases, the communication among function blocks becomes the new system performance bottleneck [1].

The simplest way of connecting the multiple function blocks on a single chip is to use a traditional system bus. But, the existing buses may not be the solution to the communication traffic problem because only one pair of master and slave blocks can send and receive data at a particular time. There are several types of on-chip bus proposals like AMBA [2] from ARM, CoreConnect [3] from IBM, WISHBONE [4] and etc. to resolve this problem. However, they are also limited in the sense that they do not deal with communications among multiple buses on a single chip.

Bus segmentation that was proposed to reduce the average power to drive the long bus lines may contribute to the performance improvement with concurrent bus transactions [5]. However, the approach has limited concurrency due to the inherent nature of the segmented bus. To improve system throughput further, we propose a new two-level bus interconnection scheme. Our approach allows concurrent bus transactions

P.-C. Yew and J. Xue (Eds.): ACSAC 2004, LNCS 3189, pp. 363–372, 2004.

for a multiple bus clusters. We assume that the modules on a chip are clustered according to their traffic patterns. Our bus system is constructed in a two-level manner to separate local traffic from inter-bus traffic that is dealt with inter-bus connection paths. Our inter-bus connection scheme employs multiplexers rather than tri-state buffers or switches for reliable bus system operations and higher testability. With performance simulations, we observed that our MUX-based two-level on-chip bus architecture improves bus throughput by about 1.5 times compared with the existing segmented bus architecture. We also noticed from our preliminary implementations that our approach is more effective in reducing clock period than the segmented bus approach.

In the next section, previous works on the segmented bus were described and analyzed. In section 3, we describe our two-level bus architecture with several configurations. In section 4, we compare our bus structure with the existing segmented on-chip bus structure based on performance simulation. Finally, we summarize this paper and suggest future works.

2 Segmented On-Chip Bus System and Its Limitations

Several works on bus segmentation were proposed to reduce the average power consumption to drive long bus lines [6, 7]. In addition, the approach achieves throughput enhancement through parallelism [5]. The core idea of the existing segmented on-chip bus approach is to partition a bus system into several bus segments considering traffic localization. Each segment is a normal bus consisting of masters and slaves. Fig. 1 shows the segmented bus organization with four segments. Adjacent segments are connected via switches or tri-state buffers. The internal bus transactions of different segments can be performed concurrently while the switches connecting the segments are turned off. For example, four concurrent intra-segment transactions can be performed in the bus organization shown in Fig. 1 while all the switches connecting the segments are off. In addition, an inter-segment transaction between two different segments can also be performed through the switches connecting the corresponding segments. Multiple inter-segment transactions may be performed concurrently if the data paths of the transactions do not overlap. For example, we may perform the transaction between segments #1 and #2 and the transaction between segments #3 and #4 concurrently in the bus structure shown in Fig. 1.

The switches are controlled by a central arbiter that receives request signals and target segment identifiers from local arbiters. To implement the segmented bus structure, only arbiters need to be modified and the inter-segment transaction is transparent to all the function blocks in the system. Function blocks need not to be modified to allow inter-segment transactions on the segmented bus. In Fig. 1, CA represents central arbiter and LA local arbiter. LA selects one of transaction requests based on its own arbitration policy. If the transaction is an inter-segment, LA sends request to CA and CA sends acknowledge to LA when the inter-segment transaction is ready to start. CA has its own arbitration algorithm to select one transaction if several local arbiters ask more than one inter-segment transactions at a time.

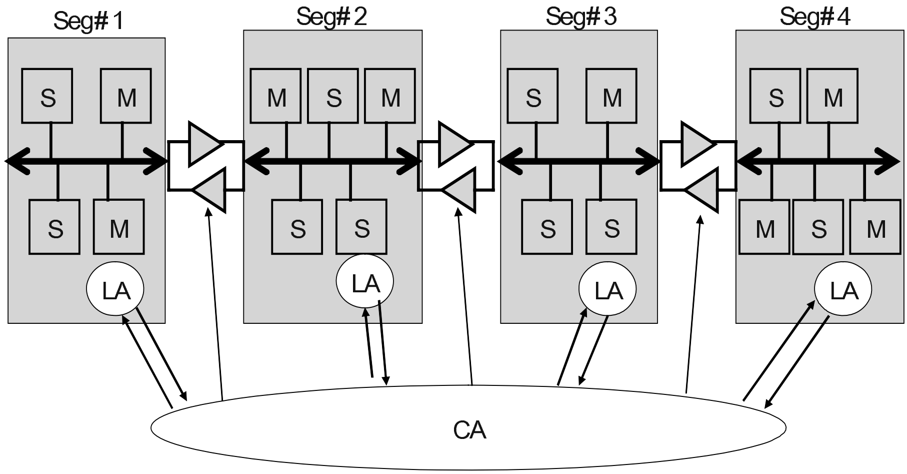


Fig. 1. Segmented Bus Organization

Though the segmented bus is beneficial in many aspects, it has several limitations as follows.

1. If several adjacent segments are occupied with an inter-segment bus transaction, other transactions (including inter- or intra- segment transactions) cannot be performed at the moment. It is because the related segments form a single bus that allows only one transaction at a time. For example, in Fig. 1, while segment #1 (master) sends data to segment #3 (slave), segment#2 cannot start any bus transaction until the inter-segment transaction between segments #1 and #2 finishes.

2. The effect of energy saving by the capacitive load reduction in the segmented bus may be alleviated by unnecessary power consumption due to tri-state buffers. Note that the undefined value (tri-state may result in an undefined value) of an input signal for CMOS logic makes a short circuit path between VDD and GND [9]. Such an undefined value also makes the logic testing difficult since undefined signal values prohibit the exact probing of logic values.

3. The clock period of the segmented bus system increases linearly as the number of segments increases, since all the bus segments need to be driven together in the worst case.

Thus, we proposed a new on-chip bus scheme that further improves parallelism while guaranteeing no undefined state and reducing clock period. Our approach is a two-level bus system using multiplexers instead of tri-state buffers in order to avoid adverse effects.

3 Proposed MUX-Based Two-Level Bus System

The proposed two-level MUX-based bus organization consists of local independent buses that contain sets of closely communicating modules, multiplexers that route inter-bus transactions between local buses, and a central arbiter that controls the multiplexer(s) for inter-bus connection at a higher level. Fig. 2 shows a simple MUX-based bus organization with only one multiplexer set. In that case only one inter-bus transaction can occur at a time while multiple local bus (intra-bus) transactions can be performed concurrently in the local buses that are not involved in the current inter-bus transaction.

Such a bus configuration has higher concurrency than the segmented bus because the local buses that are not involved in an inter-bus transaction need not suspend as in the case of the segmented bus organization. In addition, we can avoid unnecessary power consumption by employing multiplexers rather than tri-state buffers. If more than two inter-bus transactions are requested simultaneously the arbiter selects one of them by an arbitration algorithm and controls the local arbiters and the multiplexers accordingly.

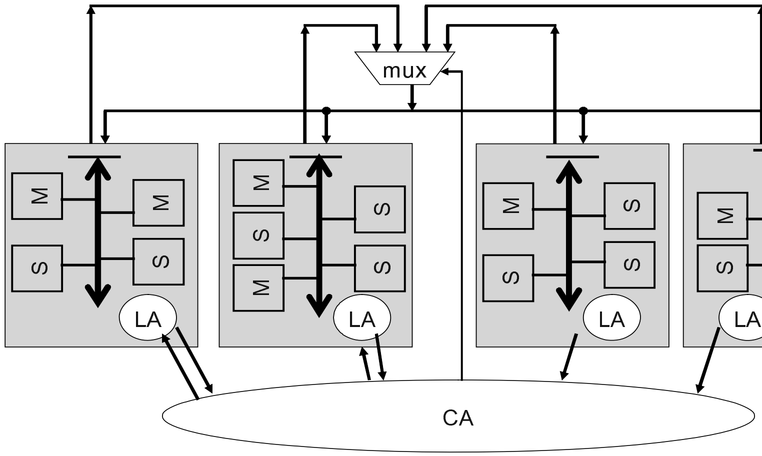


Fig. 2. A simple MUX-based bus organization

We can try another bus configuration and obtain more concurrency if we add more multiplexers as shown in Fig. 3 and Fig. 4. The bus configuration shown in Fig. 3 uses two 4-to-1 multiplexers and four 2-to-1 multiplexers to allow two concurrent inter-bus transactions. Since up to two inter-bus transactions may occur simultaneously in this configuration, we may expect more inter-bus transaction concurrency than in the bus configuration with one multiplexer. If more than two buses request for the same target local bus simultaneously, central arbiter selects one of them according to an arbitration policy.

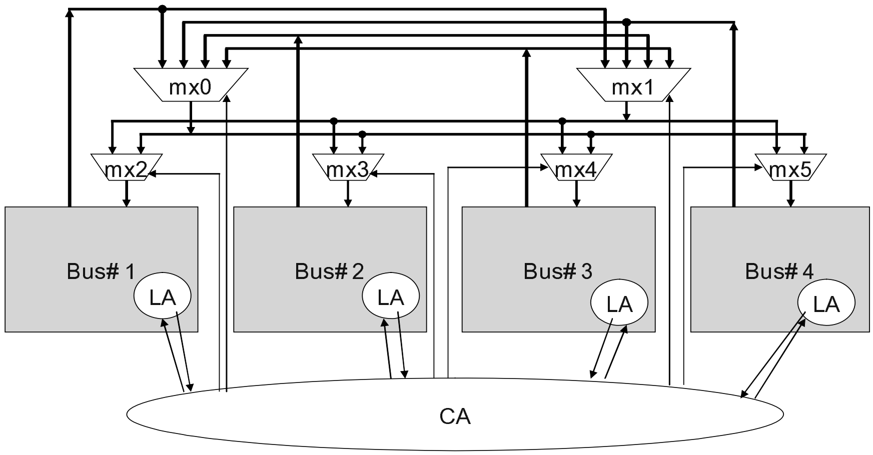


Fig. 3. MUX-based bus organization with two concurrent inter-bus transactions

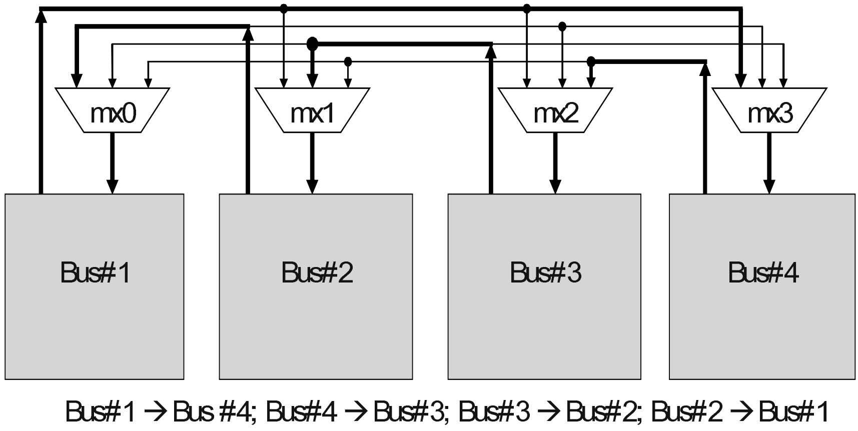


Fig. 4. Four concurrent inter-bus transactions in a structure with four 3-to-1 multiplexers

We can maximize inter-bus concurrency with four 3-to-1 multiplexers as shown in Fig. 4. It is possible for such a bus configuration to have up to four concurrent inter-bus transactions because the master part and the slave part in a local bus may be involved in different inter-bus transactions. For example, the master part of Bus#1 sends the data to the slave part of Bus#2 while the master part of Bus#2 sends data to the slave part of Bus#1 simultaneously. Fig. 4 shows an example of four concurrent inter-bus transactions.

Our approach is similar to the segmented bus approach with a central arbiter and local arbiters. But, our approach is different in that the master part and the slave part in a local bus can be involved in different inter-bus transactions resulting in maximal inter-bus transaction concurrency.

Note that we can use the existing modules without modification as is the case with segmented bus because the inter-bus transaction is transparent to each modules involved. In addition, our bus architecture has higher possibility to achieve more concurrency than the segmented bus especially when the local buses are also MUX-based. The local on-chip bus may be a bi-directional bus that use one bus line for both input and output data or a MUX-based bus that separates data-in and data-out bus lines from each other. However, in order to achieve the maximum bus throughput, we assume that the local buses are also multiplexer-based bus system.

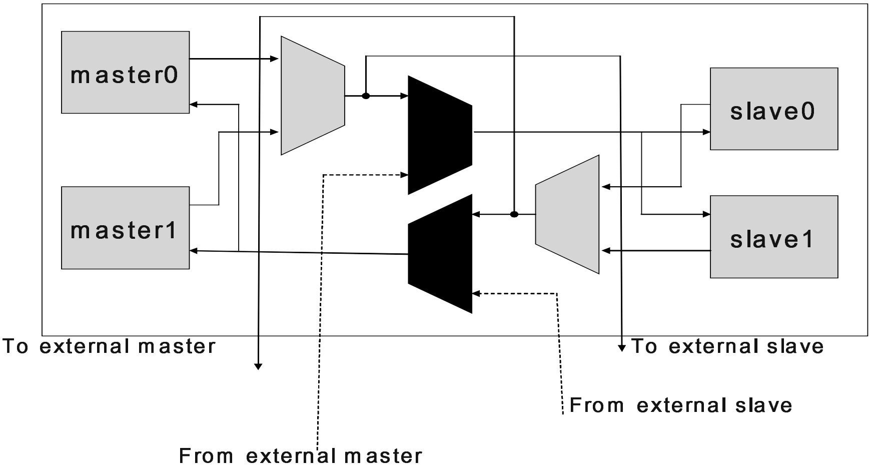


Fig. 5. An internal structure for the multiplexer-based local bus

Fig. 5 shows the internal bus structure that can be used with our two-level bus to maximize inter-bus transaction parallelism. The upper black MUX routed to the internal slave is used to select one from internal and external master requests while the lower black MUX routed to the internal master is to choose one from internal and external slave responses. The most important benefit of this structure is that the master part and the slave part in a bus can be involved in different inter-bus transactions, resulting in maximum inter-bus transaction concurrency.

4 Experiments and Analysis

We did the performance simulations for each bus configuration shown in the previous sections. The bus systems were modeled with VHDL in the behavioral level and simulated by ModelSim II simulator to measure the bus system performance. In the models we assumed that each bus transaction has the same length and target addresses are generated based on uniform distribution random number function. Each local arbiter selects one master if there is any intra-bus request. Or, the local arbiter selects one of masters requesting inter-bus transactions and forwards the request to the central arbiter. Central arbiter selects and grants one request for the same target bus based on a round-robin policy. In experiments, we have two adjustable parameters for each local bus: (1) idle time ratio (*idle_ratio*) that indicates how much portion of the time the local bus is idle (2) inter-bus or inter-segment transaction ratio (*inter_bus_ratio*) over the whole active bus transactions. So, the time spent for the inter-segment or inter-bus traffic can be calculated by the formula, (1- *idle_ratio*) * *inter_bus_ratio* * simulation_time. Our simulation assumes the number of segments or local buses is four and we deal with the following four types of bus configurations. The number of local buses can be larger than four to include more components in a system.

- The existing segmented bus of Fig. 1 (*segmented*)
- The MUX-based two-level bus with one multiplexer of Fig. 2 (*mux1)*
- The MUX-based two-level bus with two multiplexer of Fig. 3 (*mux2*)
- The MUX-based two-level bus with four multiplexer of Fig. 4 (*mux4*)

Fig. 6, Fig. 7 and Fig. 8 are graphical representations of the performance simulation results for the varying idle ratio (10%, 30% and 50% respectively) and inter-bus transaction ratio (X-axis). Y-axis indicates throughput, i.e. the time spent for transactions (inter-bus transactions + intra-bus transactions) divided by the total simulation time. We can see from the figures that the maximum throughput decreases as *idle_ratio* increases (Note the scales of Y-axis from Figure 6 to Figure 8). In addition, throughput goes down with the increase of *inter_bus_ratio* regardless of *idle_ratio*. This implies that when *inter_bus_ratio* is high, throughput is limited largely by the number of MUXes dedicated to inter-bus connections.

From the simulation results, we can confirm that *mux4* is better than any other bus configurations. The simulation graphs indicate that the structures, *segmented*, *mux1*, *mux2* and *mux4* can be ordered by the system throughput as follows.

mux 4 > *mux2* > *mux1* > *segmented*

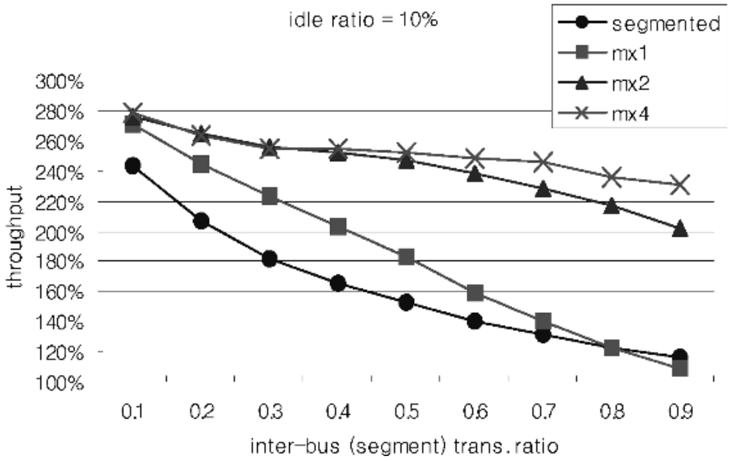


Fig. 6. Graphical Representation of Simulation Results for Idle Ratio=10%

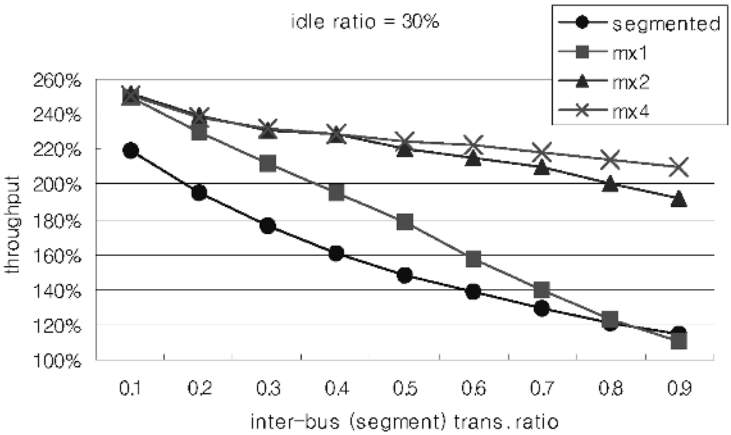


Fig. 7. Graphical Representation of Simulation Results for Idle Ratio=30%

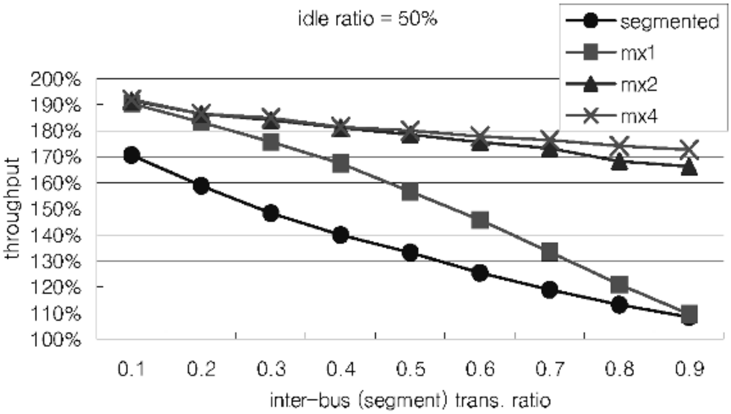


Fig. 8. Graphical Representation of Simulation Results for Idle Ratio=50%

Especially, *mux4* shows higher performance than *segmented* by 1.2 to 1.5 times. In addition, we observe that the slope of *mux4* configuration less steep than *mux1* and *segmented*. That means *mux4* configuration is less sensitive to the variation of the inter-bus transaction ratio than others. We can state that *mux4* bus configuration provides relatively stable bus traffic quality compared with other on-chip bus configurations for the fluctuating inter-bus transaction ratio.

For high inter-bus transaction ratio (0.9 or 0.8), *segmented* is slightly better for *mux1*. This is because *mux1* allow only one inter-bus transaction at a time, while *segmented* permits two concurrent inter-bus transactions at a time. Note that higher inter-bus (segment) ratio means most bus transactions are of inter-bus or inter-segment traffic. Remember that since clustering is made reflecting the traffic locality, the likelihood of high inter-bus transaction ratio is so small. Therefore, we can consider that *mux1* is better choice than *segmented* in a normal situation.

During the simulation, we assumed that the arbitration decision of each local arbiter and that of the central arbiter are independent. Thus, the selection of a master in each local bus is determined based on just the local information of the bus. Such a policy is so simple that it is easy to implement, but the policy has limited efficiency for the maximal throughput of the whole system. We suppose that *mux4* is just slightly better than *mux2* (especially for low inter-bus transaction ratio) due to this simple arbitration policy.

Clock period is another factor to determine the system performance in a synchronous system. We analyzed the circuit components affecting clock periods of the aforementioned configurations. Our analysis is summarized in Table 1. The configuration *segmented* seems to have more delay components than others since four separate buses must act like a single bus in the worst case. In that case, the bus system should drive four internal bus wires and three switches in a clock cycle. On the other hand, MUX-based bus configurations have similar delay factors to determine clock period regardless of the number of MUXes employed.

Table 1. Major components that affect clock period of each configuration.

Segmented	*mux1*	*mux2*	*mux4*
4 bus wires + 3 switches	1 bus wire + 1 int. MUX + 1 ext. MUX + ext. bus wire	1 bus wire + 1 int. MUX + 2 ext. MUX + ext. bus wire	1 bus wire + 1 int. MUX + 1 ext. MUX + ext. bus wire

We implemented partially our proposed architecture (*mux2*) and the segmented bus structure with synthesizable RTL VHDL targeting XILINX FPGA (XCV3000). Synthesis results with XILINX design tool (ISE 6.2) show that *segmented* configuration has about 2.5 times longer clock period than *mux2* configuration. On area and power consumption, two configurations exhibit similar results. This indicates that our approach is feasible and viable.

5 Summary and Future Works

As the system integration level gets higher, on-chip bus interconnection between many IP modules is required for higher performance through exploiting the bus transaction parallelism. In this paper, we propose a new on-chip bus interconnection architecture that could maximize parallelism among buses or segments. We compare the performance of our proposal with the existing segmented on chip bus proposal. Our idea is based on the two-level bus system with multiplexers rather than tri-state buffers or switches that cause some power consumption and testability problems. Our on-chip bus system has more stability and testability as well as higher parallelism than the segmented bus system. The performance simulation results show that our interconnect method has higher system performance by 1.5 times than that of existing segmented bus approach. We can summarize the performance simulation results as follows.

1. Generally, the system performance relation is *mux4 > mux2 > mux1 > segmented*
2. The bus configurations *mux4* and *mux2* show more stable traffic quality than *mux1* and *segmented* bus configuration for varying inter-bus transaction ratio.

The simulation result of our MUX-based bus (*mux4*) that allows four concurrent inter-bus transactions was slightly better than our MUX-based bus (*mux2*) that allows two concurrent inter-bus transactions due to the trivial arbitration policy. This implies a further study on the arbitration algorithm is necessary to maximally exploit the topological benefits of MUX-based bus system.

Our preliminary implementation on *segmented* and *mux2* shows that the clock period of *segmented* configuration is about 2.5 times longer than that of our proposed configuration *mux2* while both approaches exhibit similar area and power consumption. This indicates that our approach is feasible and viable.

We feel the necessity to adapt the idea to the popular existing MUX-based on chip bus architectures like AMBA. Currently, we are working towards to hardware implementation of the proposed MUX-based two-level bus system. Finally, we hope to apply our bus system to the real world examples such as multimedia processing application.

Acknowledgements. This work was supported in part by IT SoC Promotion Group of Korean IT Industry Promotion Agency and partly by ETRI (Electronics and Communications Research Institute) in Korea.

References

1. D. Langen, A. Brinkmann, and U. Ruckert, “High Level Estimation of the Area and Power Consumption of On-Chip Interconnects”, Proc. of the 13th Annual IEEE International ASIC/SOC Conference, Sep. 2000. pp. 297-301.
2. “AMBA 2.0 Specification,” http://www.arm.com/products/solutions/AMBA_Spec.html
3. “The CoreConnect Bus Architecture,”
http://www-3.ibm.com/chips/products/coreconnect/
4. “Wishbone,” http://www.opencores.org
5. T. Seceleanu, J. Plosila, and P. Liljeberg, "On-Chip Segmented Bus: A Self-timed approach",IEEE ASIC SoC Conference 2002.
6. C.-T. Hsieh and M. Pedram, "Architectural energy optimization by bus splitting," IEEE Tran. CAD, vol. 21, no.4, APR. 2002.
7. J.Y. Chen, W. B. Jone, J. S. Wang, H.-I. Lu, and T. F. Chen, "Segmented bus design for low-power systems" IEEE Tran. VLSI Systems, vol. 7, no.1, Mar. 1999.
8. K. S. Jhang, K. Yi, and Y. S. Han, “An Efficient Switch Structure for Segmented On-Chip Bus”, Proc. of Asia-Pacific International Symposium on Information Technology, Jan. 2004.
9. Gary Yeap, Practical Low Power Digital VLSI Design, Kluwer Academic Publisher, 1998.

Make Computers Cheaper and Simpler

GuoJie Li

Institute of Computing Technology
Chinese Academy of Sciences

Abstract. It is expected that 700-800 millions of Chinese people will be connected to Internet in the next 15 years. What is really needed is reducing the cost of computers so that information services will be available for low income people rather than high performance only. In this talk, we briefly examine the research and development of computer architecture during the past decades and rethink the Moore's Law from the user's point of view. To explain the design principle of low cost and simplicity, we discuss the prospects of the new research direction by using examples from our own work, such as low cost CPU design, massive cluster computer (MCC) and reconfigurable computer system. Our research shows that it is possible to design and implement the deskside Teraflops-level supercomputer, which is less than $100K, and PC of $150 in the next 3-5 years.

P.-C. Yew and J. Xue (Eds.): ACSAC 2004, LNCS 3189, p. 373, 2004.

A Low Power Branch Predictor to Selectively Access the BTB

Sung Woo Chung and Sung Bae Park

Processor Architecture Lab., Samsung Electronics,
Giheung-Eup, Yongin-Si, Gyeonggi-Do, 449-711 Korea
{s.w.chung, sung.park}@samsung.com

Abstract. As the pipeline length increases, the accuracy in a branch prediction gets critical to overall performance. In designing a branch predictor, in addition to accuracy, microarchitects should consider power consumption, especially in embedded processors. In this paper, we propose a low power branch predictor, which is based on the gshare predictor, by accessing the BTB (Branch Target Buffer) only when the prediction from the PHT (Prediction History Table) is taken. To enable this, the PHT is accessed one cycle earlier to prevent the additional delay. As a side effect, two predictions from the PHT are obtained at one access to the PHT, which leads to more power reduction. The proposed branch predictor reduces the power consumption, not requiring any additional storage arrays, not incurring additional delay (except just one MUX delay) and never harming accuracy. The simulation results show that the proposed predictor reduces the power consumption by 43-52%.

1 Introduction

As the pipeline length of today's embedded processors increases, the accuracy of a branch prediction affects the performance more significantly. In addition to accuracy, processor architects should consider the power consumption in a branch predictor, which is reported to account for more than 10% of the total processor's power consumption [1]. (In this paper, we define that a branch predictor is composed of a PHT (Prediction History Table) and a BTB (Branch Target Buffer)) Especially, in the embedded processor where the power consumption is crucial, it is important to reduce the power consumption of the branch predictor while maintaining the accuracy.

Some general purpose processors exploit the predecoding to detect whether the instruction is a branch or not, resulting in the selective access to the branch predictor(PHT and BTB) only when the instruction is a branch. In this case, the access to the branch predictor should be preceded by the access to the instruction cache. However, if the fetch stage is timing critical, the sequential access incurs additional delay. In embedded processors such as ARM 1136 [8] and ARM 1156 [9], the fetch stage is timing critical, leading to the simultaneous accesses to the branch predictor and the instruction cache. Accordingly the branch predictor

P.-C. Yew and J. Xue (Eds.): ACSAC 2004, LNCS 3189, pp. 374–384, 2004.

should be accessed whenever instructions are fetched, resulting in significant power consumption.

Note that all the predictions from the branch predictor are not used to fetch the next instructions. In some processors that have timing margin in the fetch stage, instructions are predecoded after instructions are fetched. If the instruction is predecoded as a branch, the prediction is used to fetch the next instruction. Otherwise, the prediction is not used and the instruction of subsequent address is fetched. In other processors that have no timing margin in the fetch stage, the predictions are used to fetch the next instruction only when there is a hit in the BTB. In other words, a BTB hit is considered to indicate that the instruction is a branch.

Banking could be considered to reduce the power consumption in the branch predictor. However, the power reduction by banking is not more than 4% [1] and banking requires additional chip area. The PPD(Prediction Probe Detector), which is accessed earlier than the instruction cache, was proposed for a low power branch predictor [1]. It detects whether the instruction is a branch instruction in the fetch (branch prediction) stage to eliminate unnecessary accesses to the branch predictor. However, the PPD itself consumes extra power. Moreover, it may increase the pipeline latency, which may lead to the decrease the processor frequency [2][3].

In this paper, we propose a low power branch predictor. In the proposed branch predictor, the BTB is accessed only when the branch prediction is taken. To enable this without additional delay, the PHT is accessed one cycle earlier, resulting in selective access to the BTB.

2 Branch Predictors for Embedded Processors

Static branch predictors have been used for embedded processors. In some embedded processors, bimodal branch predictors, which are known as moderately accurate, are adopted. Recently, however, the increased pipeline length of embedded processors causes more performance penalty from branch mispredictions. Accordingly, a more accurate branch prediction is necessary to reduce the CPI loss that arises from the longer pipeline. Considering only accuracy, the tournament branch predictor [4], which was used for Alpha 21264, would be one of the best choices. However, it requires too large chip area to be adopted in embedded processors. The accurate branch predictor, which consumes reasonable area and power, such as gshare [4], is being considered for embedded processors [9].

Fig. 1 depicts the PHT of the traditional branch predictor, called gshare [4]. The PHT is an array of 2-bit saturating counters. It is indexed by the exclusive OR of the PC with the global history. Each counter of the PHT is increased/decreased when the branch is taken/untaken. The MSB (Most Significant Bit) of each entry determines the prediction (branch taken/untaken).

The branch predictor inevitably consumes unnecessary power. As shown in Fig. 2, whenever there is an instruction in the fetch stage, the PHT and the BTB should be accessed simultaneously. The reason is that there is no way to detect

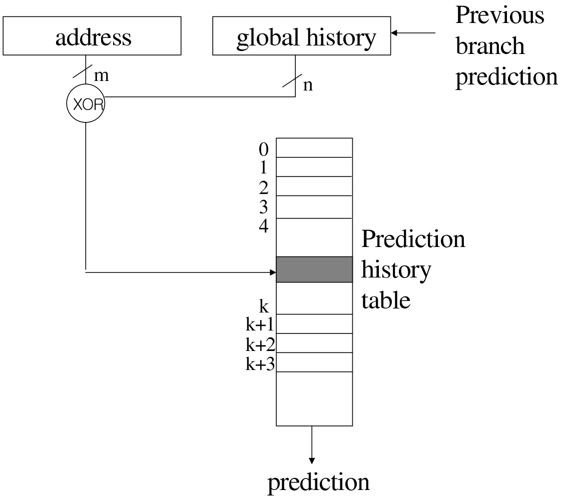


Fig. 1. The PHT of the Traditional Branch Predictor (Gshare Branch Predictor)

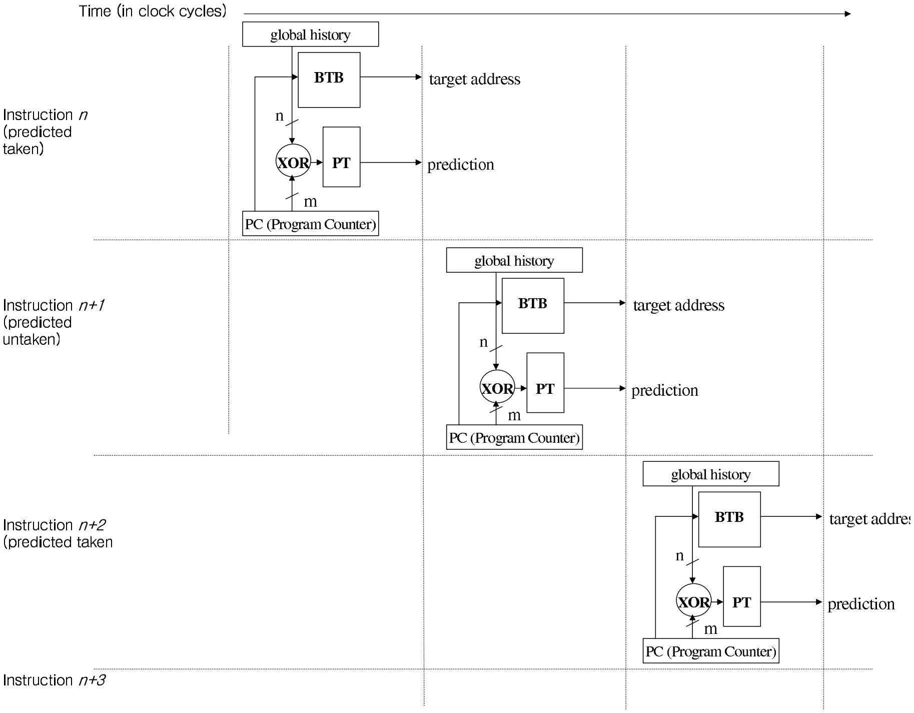


Fig. 2. Branch Prediction in the Traditional Branch Predictor

whether the instruction is a branch or not, in the early part of the fetch (branch prediction) stage. If the instruction is a branch, the PHT will be reaccessed later for training (update of the prediction information). Otherwise, there is no need to update the PHT for training.

3 Low Power Branch Predictor

In the proposed predictor, the BTB is accessed only when the prediction from the PHT is taken. To avoid the additional delay, the PHT should be accessed one cycle earlier.

3.1 Early Access to the PHT

In the traditional predictor, the PHT is accessed every cycle when there is a fetched instruction. To reduce the power consumption in the PHT, the proposed predictor looks up two predictions at every access to the PHT, which is described in Fig. 3. We double the width of the PHT but reduce the depth of the PHT by half for a fair comparison. The dynamic power consumption of the 4096 X 2 PHT is more than that of the 2048 X 4 PHT, though the difference is just 4.2 % - This data is obtained using Samsung Memory Compiler [5].

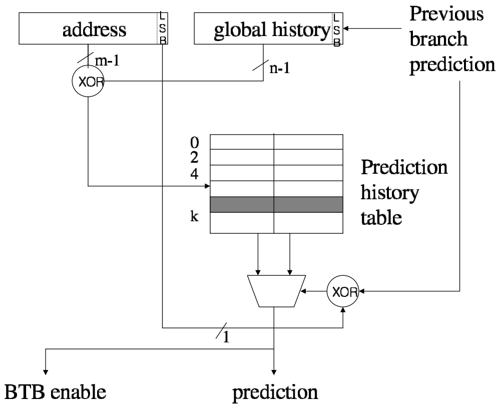


Fig. 3. Branch The PHT of the Proposed Branch Predictor

Different from the traditional predictor, the PHT in the proposed predictor is indexed by exclusive ORing of the PC excluding the LSB with the global history excluding the LSB. Note that if the global history is not changed and only the LSB of the PC is changed, two predictions can be acquired by accessing the PHT only once. The proposed predictor assumes that the previous instruction is not a predicted taken branch which changes global history. In other words, the propose predictor assumes that there is always sequential access to the PHT. If the previous instruction is a predicted taken branch, the PHT is reaccessed with new PC and new global history.

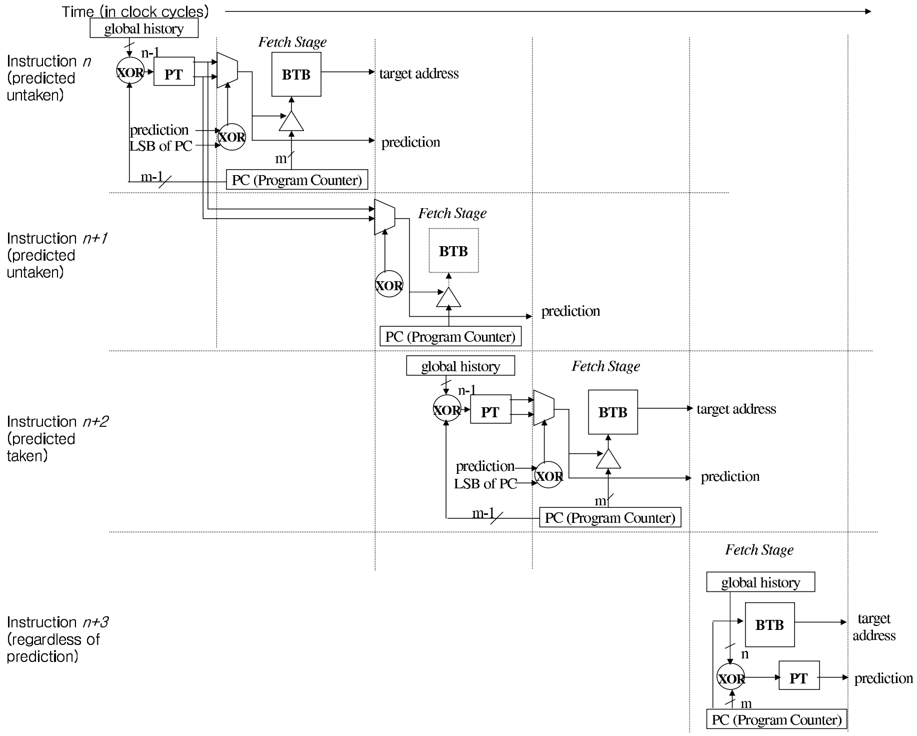


Fig. 4. Branch Prediction in the Proposed Branch Predictor

As shown in Fig. 4, the PHT lookup is done one cycle earlier compared to the traditional predictor. After the PHT lookup, the prediction is selected between the two predictors in the fetch (branch prediction) stage. Thus, the prediction can be obtained just after the MUX delay, which is early in the fetch stage compared to the traditional predictor (ex. in case of instruction n and instruction $n + 1$ in Fig. 4). If the previous instruction (instruction $n + 2$ in Fig. 4) is predicted as taken, the PHT should be reaccessed with a new PC in the fetch stage for the current instruction (instruction $n + 3$ in Fig.4). Note that two predictions can be used only when they are correctly aligned. For example, predictions for instruction n and instruction $n + 1$ can be looked up by accessing the PHT just once, whereas predictions for instruction $n + 3$ and instruction $n + 4$ can not since they are misaligned in PHT as shown in Fig. 3.

If the PHT were sequentially accessed, the number of accesses to the PHT would be decreased by 50% in the proposed predictor. Practically, however, the PHT is not always sequentially accessed because branch instructions, regardless of taken/untaken, change the global history (If the branch is predicted taken, the PC is also changed). Accordingly, the number of accesses to the PHT is decreased to ((total number of instructions)/2 + number of branch instructions). Generally, branch instructions occupies only 0-30% of total instructions in most applications [6][7], resulting in substantial power reduction.

As explained above, the power consumption is reduced. How much is the accuracy of the proposed predictor affected? If the previous instruction is predicted untaken, the prediction for the current instruction is also sequentially accessed, which does not make any difference. Thus, the prediction from the proposed predictor is same as that from the traditional predictor. If the previous instruction is predicted taken, the PHT is reaccessed in the fetch stage for the predicted address from the BTB. Though the prediction from the first access to the PHT is different, the prediction from the second access to the PHT of the proposed predictor is same as that of the traditional predictor. Therefore, the proposed predictor never affects the accuracy of the branch prediction.

3.2 Selective Access to the BTB

In the traditional predictor, the BTB as well as the PHT should be accessed every cycle. As mentioned in Sect. 3.1, the prediction is known early in the fetch stage of the proposed branch predictor, which enables the proposed predictor to selectively access to the BTB, as shown in Fig. 4. If the prediction is taken, the access to the BTB is enabled. Otherwise, the access to the BTB is disabled because the target address, which is obtained from the BTB, is useless. Though the number of predicted takens is generally more than that of predicted untakens, the predicted untakens still occupy 10-60% of total instructions, depending on the applications [6][7]. In other words, 10-60% of the BTB accesses can be decreased, resulting in the power reduction of the BTB.

4 Analysis Methodology

We presented analytical model and ran the simulations to evaluate the power consumption. Simulations were performed on a modified version of Simplescalar toolset [10]. The power parameters were obtained from the Samsung Memory Compiler [5] with Samsung 0.13 um generic process in a typical condition (25^{o}C, VDD=1.20V). The configuration of the simulated processor is based on the specification of ARM 1136 [8]. We selected four applications (gcc, mcf, perl_bmk and vortex) from the Spec2000 benchmark suite [6]. Table 1 shows the sizes of the PHT and the BTB, which are expected for embedded processors in the near future. The power consumptions in Table 1 are normalized to the power consumption in the case of the power consumption of the read operation for 4096 X 2 PHT (We normalized the value since it is not permitted to officially present the absolute value by Samsung internal regulation).

Table 1. Size and normalized power consumption of components

Component	Type	Size	READ	WRITE
PHT	Traditional Predictor	4096 X 2	1.00	0.91
PHT	Proposed Predictor	2048 X 4	1.04	0.99
BTB	Traditional/Proposed Predictor	256 entry	1.88	2.14

Table 2. Notations for the analytical model

Notation	Meaning
P(trad.)	Total power consumption in the traditional predictor
P(prop.)	Total power consumption in the proposed predictor
$P_{pht\_traditional\_read}$	Power consumption of a read operation in the traditional PHT
$P_{pht\_traditional\_write}$	Power consumption of a write operation in the traditional PHT
$P_{pht\_proposed\_read}$	Power consumption of a read operation in the proposed PHT
$P_{pht\_proposed\_write}$	Power consumption of a write operation in the proposed PHT
P_{btb_read}	Power consumption of a read operation in the BTB
P_{btb_write}	Power consumption of a write operation in the BTB
N_{inst}	Number of total instructions
N_{branch}	Number of branch instructions
$N_{btb\_miss\_predictions}$	Number of BTB miss predictions for branch instructions
$N_{untaken\_predictions}$	Number of predicted untakens

5 Analysis Results

5.1 Analytical Models

We present the notations in Table 2 for analytical models.

The total power consumption in the traditional branch predictor is

$$\mathrm{P(trad.)} = P_{pht\_traditional\_read} \ \mathrm{X} \ N_{inst} + P_{pht\_traditional\_write} \ \mathrm{X} \ N_{branch} + P_{btb\_read} \ \mathrm{X} \ N_{inst} + P_{btb\_write} \ \mathrm{X} \ N_{btb\_miss\_predictions}$$

In the traditional branch predictor, the PHT and the BTB are read at every instruction. The PHT is written (updated) for training when the instruction is a branch and the BTB is written when a BTB miss prediction occurs.

The total power consumption in the proposed branch predictor is

$$\mathrm{P(prop.)} = P_{pht\_proposed\_read} \ \mathrm{X} \ (N_{inst}/2 + N_{branch}) + P_{pht\_proposed\_write} \ \mathrm{X} \ N_{branch} + P_{btb\_read} \ \mathrm{X} \ (N_{inst} - N_{untaken\_predictions} \ ) + P_{btb\_write} \ \mathrm{X} \ N_{btb\_miss\_predictions}$$

In the proposed branch predictor, the number of read accesses to the PHT is decreased. In addition, the number of read accesses to the BTB is decreased from total number of instructions to the number of predicted takens, since the BTB

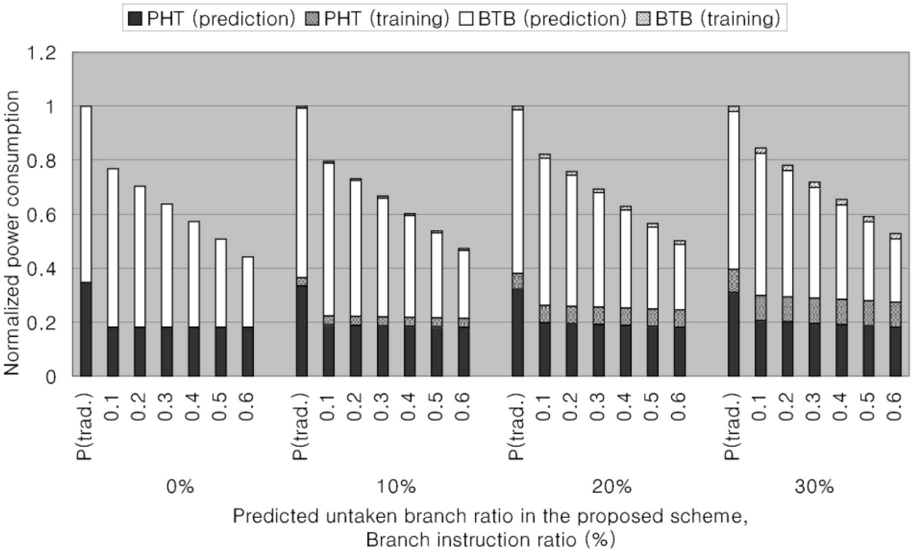


Fig. 5. Results from Analytical Models

is accesses only when the prediction from the PHT is taken. Thus the power consumption in the proposed branch predictor is expected to be reduced.

In Fig. 5, we vary the branch instruction ratio and the predicted untaken ratio to analyze the power consumption with analytical models. Each set of bars is classified, according to the branch instruction ratio in a program. In each set, all bars are normalized to the leftmost bar, which is the power consumption in the traditional predictor. As shown in above formula, as the traditional predictor is not related to the number of predicted untakens, leftmost one bar in a set is responsible for the traditional predictor with various predicted untaken ratios. Since the branch instruction ratio is 0-30% and the predicted untaken ratio is 0.1-0.6 in most applications [6][7], we showed the results in the range. In most applications, the BTB miss prediction ratio is between 0-20% but it does not affect the analysis result directly. Thus, the BTB miss prediction ratio is fixed to 10%.

The decrease of the branch instruction ratio reduces the number of accesses to the PHT and the increase of the predicted untaken ratio reduces the number of accesses to the BTB. Thus, as the branch instruction ratio decreases and the predicted untaken ratio from the branch predictor increases, the power consumption in the proposed predictor is more reduced.

The power consumption for training is same in the traditional predictor and in the proposed predictor, since the PHT training is necessary whenever there is a branch instruction and the BTB training is necessary whenever a BTB miss prediction occurs. As shown in Fig. 5, the power consumption for prediction (not training) in the PHT and in the BTB occupies 29-35% and 55-65% of the total power consumption in the branch predictor, respectively. The proposed predictor reduces the power consumption not for training but for prediction. The power reduction from prediction in the PHT is 11-16%, depending on the branch instruction ratio. Moreover, the power reduction from prediction in the BTB is 5-39%, depending on the predicted untaken ratio. Therefore, the proposed predictor reduces the total power reduction of the branch predictor by 16-55%.

6 Simulation Results

Fig. 6 shows the simulation results with four applications from Spec2000 benchmark suite [6]. The power reduction from the PHT is 13-16%, depending on the branch instruction ratio. The power reduction from the BTB is 30-36%, depending on predicted untaken ratio. The power reduction from the overall proposed branch predictor is 43-52%. On average, the power reduction from the PHT, from the BTB and from the overall proposed branch predictor is 14%, 32%, and 46%, respectively.

Table 3 shows the branch instruction ratio and the predicted untaken ratio to compare the results from analytical models and results from simulations. If the parameters in Table 3 are put in the analytical models, the results from analytical models in Fig. 5 are similar to those from simulations in Fig. 6.

7 Conclusions

In this paper, we proposed a low power branch predictor for embedded processors by reducing the accesses to the PHT and the BTB without harming any

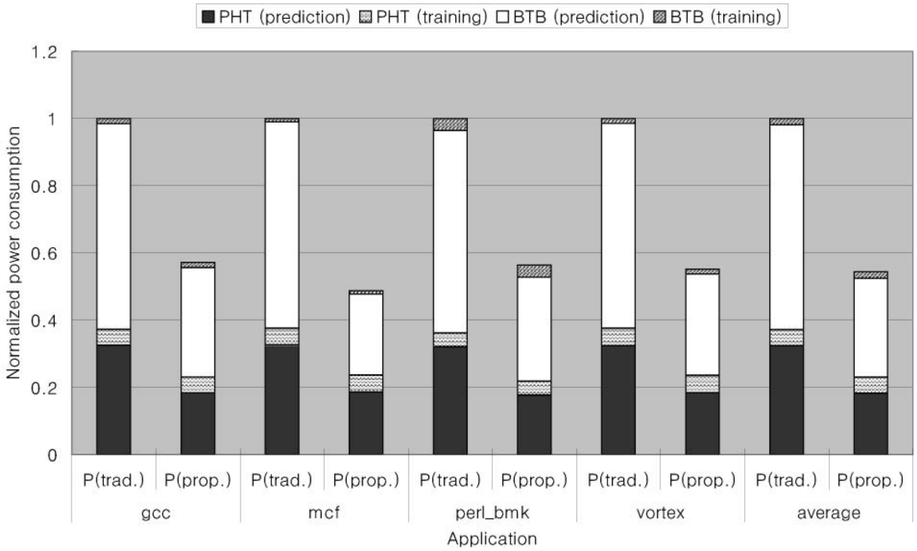


Fig. 6. Simulation Results

Table 3. Branch instruction ratio and predicted untaken ratio

Applications	Branch instruction ratio	Predicted untaken ratio
gcc	15.1 %	48.5 %
mcp	16.9 %	61.7 %
perl_bmk	14.9 %	50.0 %
vortex	13.0 %	51.5 %

accuracy: 1) Two predictions are looked up by accessing the PHT once. 2) The BTB is only accessed when the prediction from the PHT is taken, which does not incur any additional delay, since the prediction is obtained one cycle earlier compared to the traditional branch predictor.

We presented analytical models and analyzed the power consumption with the models. We also ran the simulation to investigate the power consumption in the real applications. Simulation results showed that the proposed predictor reduces the power consumption by 43-52%. Considering that the branch predictor accounts for more than 10% of overall processor power, it is expected to reduce overall processor power by about 5-7%. The proposed predictor is being considered as a next-generation embedded processor.

Acknowledgments. We would like to thank Gi Ho Park for his helpful comments; Woo Hyong Lee and for his survey of various branch predictors; Young Min Shin for his helpful comments on the 0.13 um process; Young Chul Rhee, Dong Wook Lee and Sang-Suk Kim for their assistances in setting up Samsung Memory Compiler System.

References

1. Parikh, K. Skadron, Y. Zhang, M. Barcella and M. Stan : Power issues related to branch prediction, Proc. Int. Conf. on High-Performance Computer Architecture, (2002) 233-242
2. Daniel A. Jimenez : Reconsidering complex branch predictors, Proc. Int. Conf. on High-Performance Computer Architecture (2003) 43-52
3. Daniel A. Jimenez, Stephen W. Keckler, and Calvin Lin : The impact of delay on the design of branch predictors, Proc. Int. Symp. on Microarchitecture (2000) 67-76
4. S. McFarling : Combining branch predictors, WRL Technical note TN-36, Digital (1993)
5. Samsung Electronics : Samsung Memory Compiler (2002)
6. Standard Performance Evaluation Corporation : SPEC CPU2000 Benchmarks, available at http://www.specbench.org/osg/cpu2000
7. C. Lee, M. Potkonjak, W Mangione-Smith. : MediaBench : A Tool for Evaluating Synthesizing Multimedia and Communication Systems, Proc. Int. Symp. On Microarchitecture (1997)
8. ARM Corp., ARM1136J(F)-S, available at http://www.arm.com/products/CPUs/ARM1136JF-S.html

9. ARM Corp., ARM1156T2(F)-S, available at http://www.arm.com/products/CPUs/ ARM1156T2-S.html
10. Simpelscalar LLC, The Simplescalar Tool Set 3.0 available at http://www.simplescalar.com

Static Techniques to Improve Power Efficiency of Branch Predictors

Weidong Shi, Tao Zhang, and Santosh Pande

College of computing
Georgia Institute of Technology, USA
{shiw, zhangtao, santosh}@cc.gatech.edu

Abstract. Current power-efficient designs focus on reducing the dynamic (activity-based) power consumption in a processor through different techniques. In this paper, we illustrate the application of two static techniques to reduce the activities of the branch predictor in a processor leading to its significant power reduction. We introduce the use of a static branch target buffer (BTB) that achieves the similar performance to the traditional branch target buffer but eliminates most of the state updates thus reducing the power consumption of the BTB significantly. We also introduce a correlation-based static prediction scheme into a dynamic branch predictor so that those branches that can be predicted statically or can be correlated to the previous ones will not go through normal prediction algorithm. This reduces the activities and conflicts in the branch history table (BHT). With these optimizations, the activities and conflicts of the BTB and BHT are reduced significantly and we are able to achieve a significant reduction (43.9% on average) in power consumption of the BPU without degradation in the performance.

1 Introduction

Branch prediction has a huge impact on the performance of high end processors which normally have a very deep pipeline, e.g. Pentium 4 which has 20 pipeline stages. Branch mis-prediction penalty for such a deep pipeline is very high, so it is critical to achieve accurate prediction. Many studies have been done to improve branch prediction rate by complicated designs, for example, combining several types of predictors together. Those designs often demand a significant silicon and power budget. As claimed in [5], branch predictor can potentially take up to 10% of the total processor power/energy consumption. With a new metric dimension of power, the focus becomes how to maintain the same prediction rate as a complex branch predictor but with significantly less power consumption and area.

In this paper, we propose two optimizations to reduce power consumption of the branch predictor with no degradation in the IPC or prediction accuracy but significant reduction in the branch predictor power consumption. The power consumption of a branch predictor is dominated by the large branch target buffer (BTB) and branch history table (BHT), both of which are introduced to achieve high prediction accuracy in a high-performance processor. To optimize the power consumption of branch target buffer, we introduce static branch target buffer that does not need state updates during runtime except when the program phase changes; thus activities in it are reduced

P.-C. Yew and J. Xue (Eds.): ACSAC 2004, LNCS 3189, pp. 385–398, 2004.

significantly. We use profiling to identify the branches that should reside in the branch target buffer in each program phase and preload those branches into branch target buffer when the program phase changes. The content of branch target buffer never changes during one program phase. Using static branch target buffer, we are able to maintain the performance of traditional branch target buffer at the same time eliminate most of the power consumption due to the updates to traditional branch target buffer.

To reduce power consumption of branch history table, we combine static branch prediction with hardware dynamic branch prediction. With a hybrid static and dynamic branch prediction, only branches that are hard to predict statically turn to hardware prediction, reducing branch history table activities and collisions. Such a hybrid predictor can often attain the same prediction rate as a pure hardware predictor with a much smaller branch history table and much less predictor lookups and updates, therefore consumes less power. Beyond traditional single direction static branch prediction [6, 12], we propose a hardware-assisted correlation-based static prediction scheme. This design can further improve static prediction rate and reduce hardware branch predictor overhead and power consumption. Our correlation-based static prediction encodes a branch correlation pattern into a branch instruction. Hardware assists the static prediction by providing a short 2-bit global branch history and uses it to reach a prediction according to the static correlation pattern.

The rest of the paper is organized as follows. In section 2, we present our study on power-optimized branch target buffer design and our solution, i.e., static branch target buffer. In section 3, we elaborate our correlation-based static prediction scheme. Those branches that cannot be predicted statically will turn to hardware dynamic predictor. So our predictor is a hybrid one. We describe the simulation environment and the benchmarks in section 4, then present results on the effectiveness of the two optimizations described earlier. Section 5 compares our work with previous work on static branch prediction and other recent approaches on low power branch predictors. Finally, in section 6, we conclude the paper.

2 Static Branch Target Buffer

To achieve a good address hit rate in branch target buffer, modern superscalar processors normally have a very large multi-way branch target buffer. This large buffer leads to high power consumption. Normally the power consumption of the branch target buffer takes at least 50% of the total power consumption of the branch predictor[1]. Thus, to design a power-efficient branch predictor, it is critical to reduce the power consumption of branch target buffer.

An intuitive approach to reduce the power consumption of BTB is to reduce the size of it. BTB is kept large because of two possible reasons. First, a large buffer helps reduce conflict misses. Second, a large buffer helps reduce capacity misses. If the main reason for a large BTB is the conflict misses, we can increase the associativity of the BTB. However, according to our experiments, capacity misses are major problems. Some programs have large working sets. To ensure a good address

[1] In this paper, when we talk about branch predictor, we mean both branch direction predictor and branch target buffer.

hit rate for those applications, a large BTB must be deployed. Figure 1 shows the address hit rate for six SPEC2000 benchmarks. In this paper, we will mainly study these six benchmarks because they exhibit relatively worse branch prediction performance in our experiments. The configurations are 128-entry fully-associative BTB, 256-entry fully-associative BTB, 512-set 1-way BTB, 512-set 2-way BTB and 512-set 4-way BTB. From Figure 1, for benchmarks like perl, vortex and gcc, even a 256-entry fully-associative BTB cannot achieve a comparable address hit rate to the other configurations that have much less ways but a larger number of entries. Since a fully-associative BTB does not have conflict misses, the above finding shows that some benchmarks have large working sets and complex branch behaviors, requiring a large BTB.

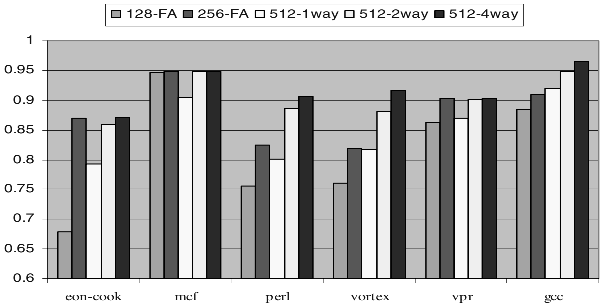


Fig. 1. Address hit rate of different BTBs

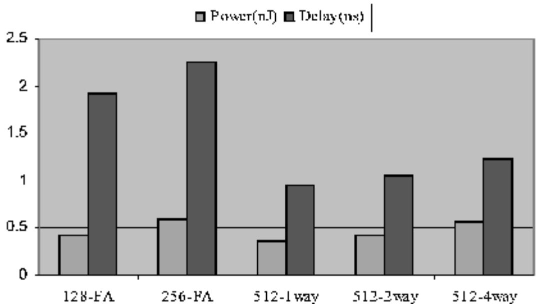


Fig. 2. Per-access Power Consumption and Delay of Different BTB Configurations

The address hit rate of fully-associative BTBs is very good for some benchmarks like mcf. For those benchmarks, we can achieve a comparable address hit rate using a fully-associative BTB with much less entries. However, Figure 2 shows per-access latency (in ns) and per-access power consumption (in nJ) of different BTBs got using CACTI timing and power model (version 3.0). From Figure 2, we can see fully-associative BTBs are not power efficient comparing to multi-way BTBs and the per-access delay of them are several times larger. Such kind of delay is intolerable to high-end processors. So the conclusion here is that we have to maintain a large enough BTB at the same time avoid introducing high associativity into the BTB.

Another way to reduce the power consumption of the BTB is to reduce its activities. With clock gating used extensively in modern processors, the power consumption of a particular function unit is significantly impacted by the activities it undertakes. Unfortunately, to reduce activities of the BTB is also a hard problem. To minimize pipeline stalls, the processor needs to know the target address of a branch as soon as possible if the branch is predicted as taken. The target address is provided by the BTB. In a traditional superscalar processor design, the processor will access BTB during the instruction fetch stage, so that the predicted target address can be fed into next fetch stage without stall. In [5], the authors proposed to reduce the activities due to BTB lookups by only accessing it when necessary. Our work is built upon the optimization proposed in [5]. We assume non-branch instructions have been filtered using a mechanism similar to the one in [5] and they will not lead to BTB lookups. Thus, our scheme can only deal with branch instructions. Except the work in [5], we are not aware of other optimizations to reduce the activities of the BTB.

In this paper, we propose static branch target buffer to reduce the activities due to BTB updates. Traditionally, whenever a branch instruction is completed, the BTB is updated to remember its target address. With the optimization in [5] enabled, BTB updates may account for half of the BTB activities. The basic idea is that if we fix the content of the BTB, then no BTB updates are necessary. A naïve implementation could be that the processor preloads BTB content when the program starts and the BTB content is never changed during the program run. This naïve implementation may work for small and simple programs but has great limitation for large and complicated programs, which may run for a long time and change behavior significantly. On the other hand, as shown in [13], although a complicated program does not exhibit a globally stable behavior, its execution can be divided into phases, in each phase the behavior is quite stable. In our static branch target buffer design, program phases are dynamically identified and program phase changes are dynamically detected in the processor. Upon a program phase change, the processor loads the BTB content corresponding to the new phase then the BTB content is fixed until next program phase change.

We use profiling to choose the proper branches to reside in the BTB for each phase. We first categorize branches encountered in the phase according to the set location it will reside in the BTB. For example, a 512-set BTB will have 512 different set locations. Normally, there will be multiple branches belonging to one set location due to collision. Next, we identify the most frequent ones belonging to a set location through profiling. The number of branches chosen are equal to the number of ways in the BTB. In that way, we choose a subset of branches which just fit into the BTB.

This idea works because program phase changes are infrequent events, otherwise, the power consumed by preloading BTB may eliminate the savings from eliminated BTB updates. We adopted the phase identification scheme from [13], which is very cost effective. Our experiment shows that GCC has the most unstable phase behavior, but on average the phases of GCC still have a length of about 1 million instructions. As pointed out by [13], integer programs tend to have much shorter program phases and much more program phase changes. For floating point programs, the length of program phases is normally tens of million instructions. All of our examined benchmarks are tough integer programs to stress test our scheme. Our static branch target buffer design will achieve even better results with floating point benchmarks.

There are several pitfalls regarding to static branch target buffer idea. Since we fix the BTB content for each phase now and it is possible that we cannot put all the branches seen in the phase into the BTB, static BTB may degrade address hit rate and runtime performance. However, sacrificing performance means the program will run for a longer time thus the other components in the processor will consume more power! Thus, although static BTB can reduce branch predictor power significantly, if the performance is degraded a lot, we may end up consume more power in the whole processor scale. Fortunately, we see near zero performance degradation under our static branch target buffer design. The major reason is that the phase identification scheme works well and captures program phases accurately. Another reason is that lots of BTB misses are actually due to some branches continuously kicking each other out of the BTB, reducing BTB effectiveness. That means fixing the content of BTB may instead help in some cases.

Static branch target buffer also introduces additional overhead to context switches. BTB content now becomes a part of process state. When a process is switched out, the current BTB of the process has to be saved. After a process is switched back, the corresponding BTB needs to be restored. However, context switches can be regarded as rare events in a processor. For example, the default time slice in Linux is 100ms, during which tens of million instructions could have been executed.

In our experiments, the power consumption of BTB preloading and phase identification has been modeled and counted in the total power consumption of the processor.

3 Correlation-Based Static Prediction

The power consumption of branch history table is another major source of branch predictor power consumption. Branch history table and branch target buffer normally take more than 95% of total branch predictor power. To reduce the power of BHT, we propose static branch prediction. The basic idea is that those branches that can be statically predicted will not go through the normal BHT lookup algorithm thus will not access BHT. So the accesses to the BHT are reduced, leading to less BHT activities thus reduced power consumption. The reduction of BHT accesses also leads to reduction of conflicts in BHT. Thus, a small history table can always achieve the same prediction rate as the traditional predictor with a much larger table. With the same BHT table, static branch prediction may help achieve a better prediction rate due to less conflicts in BHT.

The possible strategies of static branch prediction could be always predicting a branch is taken, or always not-taken, or branches with certain op code always taken, or branches with certain op code always not-taken, or always predict backward conditional branches as taken. Another approach for static branch prediction is to rely on compiler program analysis or profiling to come up with a direction hint to a conditional branch. The hint can be encoded with a single bit in the branch instruction. For example, when the bit is set, the corresponding branch is always considered as taken, otherwise not taken. Through profiling, [11] shows that a large percentage of branches are highly biased towards either taken or not-taken. In particular, 26% of conditional branches have a taken-rate of over 95% and another 36% of branches have a taken-rate below 5%. Together, over 62% of conditional branches are highly biased toward one direction. For such conditional branches, their predictions can be hard encoded and this will reduce (1) accesses to the BHT; (2) branch history table entries required for dynamic branch prediction; (3) potential conflicts and aliasing effects in BHT [4, 6]. In our approach, we further extend previous either taken or not-taken static prediction to correlation-based static prediction.

```
    if
(x>2)
        x   =
0;
    if
(y<10)
```

x>2	y<10	x<y
false	false	true
false	true	?
true	false	true
true	true	false

Fig. 3. Example of branch correlation

[12] shows that many conditional branches can be predicted by using only a short global history. This is the manifestation of branch correlation and gives the insight that many conditional branches may have fixed prediction patterns under a short global branch history. Figure 3 gives an example of branch correlation and prediction pattern.

This leads to the design of correlation-based static branch prediction. Correlation-based static branch prediction requires hardware assistance to record the recent global branch history. In many branch predictors, e.g., gshare predictors, such information is already there for dynamic branch prediction. The hardware chooses a prediction based on the current global branch history and the correlation pattern provided by the branch instruction. By using correlation-based static prediction, we can predict more conditional branches statically, which could further reduce area and power consumption of a dynamic branch predictor. The sources of power savings are listed below.

- Since many branches are predicted statically, the dynamic branch predictor can maintain the same prediction rate using a BHT with less number of entries.
- Static prediction can reduce destructive aliasing thus further alleviating the pressure on the BHT size.
- With static prediction, hardware only needs to look up BHT when a branch is not statically predicted. This reduces the number of BHT lookups. For statically predicted branches, the updates to the counters (or other finite state machines) are also eliminated, but the updates to histories are still necessary.

Static correlation pattern is conveyed to the hardware using encoding space in the displacement field of a conditional branch instruction, which is done by the compiler (binary translator). Other instructions are not affected. Conditional branch instructions in most RISC processors have a format such as [op code][disp]. For Alpha 21264 processor, the displacement field has 21 bits. However, most conditional branches are short-range branches. This leaves several bits in the displacement field that can be used for encoding correlation pattern for most conditional branches. In our correlation-based static prediction scheme, we use four bits in the displacement field (bit 4 to bit 1) to encode the prediction pattern based on two most recent branches. There are four possible histories for two branches: 1) [not-taken, not-taken], 2) [not-taken, taken], 3) [taken, not-taken], 4) [taken-taken]. Each bit of the four bits corresponds to the static prediction result for one possibility. Bit 1 corresponds to case 1, and so on. If we statically predict the branch taken, the corresponding bit is set to 1, otherwise, it is set to 0. Using the same example in Figure 3, the encoded correlation pattern for the third branch could be 0101, assume through profiling, we found if the branch history is [not-taken, taken], the third branch has a better chance to be not-taken.

Compiler has the freedom to choose whether correlation-based static prediction should be used. When it becomes unwise to use static prediction, e.g., hard to predict branches, or the displacement value is too large, compiler can always turn back to original branch prediction scheme. In our scheme, we take another bit (bit 0) of displacement field to distinguish extended branches with correlation information from original branches. Thus, besides the four bits of correlation information, we take five encoding bits in total from displacement field for extended branches. Since bit 0 of displacement field of original branches is used in our scheme, the compiler may have to transform an original branch to chained branches due to reduced displacement range, which is really rare.

First, we have to decide which conditional branches should use static branch prediction and which should use dynamic branch prediction. A conditional branch is classified as using static prediction if it satisfies the following criteria:

- highly biased towards one direction (taken or not-taken) under at least one correlation path (branch history). We maintain a history of two most recent branches in our scheme.
- branch target address's displacement within the range permitted by the shortened conditional branch displacement field. For alpha, the remaining width of displacement fields is 16 bits (21-5). The displacement permitted is [-32768, 32767]. According to our experiments, most of conditional branches (above 99.9%) are satisfied.

To evaluate the potential gains offered by correlation-based static prediction, we divide statically predictable conditional branches into three types:

- Non-correlation based type. This corresponds to conditional branches that can be statically predicted using a single-bit direction prediction. A conditional branch is classified into this type if all the program paths (histories) yield the same biased prediction, taken or not-taken.
- Correlation-based type. For this type of branches, prediction is biased towards different direction depending on the correlation path. For example, under one path (history), the branch may be biased towards taken, while under another path (history), it is biased towards not-taken.
- Others. This type corresponds to the branches that cannot be categorized to type I or type II. These branches are hard to be predicted statically.

Table 1 lists the total number of conditional branches for each type, and the total number of dynamic executions made by the branches in each type for six SPEC2000 benchmarks running for 200 million instructions with fast-forwarding 1 billion instructions. As shown in the table, many branches can be predicted statically.

One pitfall of encoding correlation information into the branch instruction is access timing. As discussed earlier, to achieve best throughput, the processor has to know the next PC address for instruction fetch at the end of current instruction fetch stage. Traditionally, during instruction fetch stage, the processor has no access to specific bits of the fetched instruction. To enable correlation-based static prediction, we use a separate extended branch information table (EBIT) with a number of entries exactly corresponding to the level-1 I-cache cache lines. Under our processor model, cache

Table 1. Conditional Branch Categorization

	# static branch	# dynamic branch	type 1 static (%)	type 1 dyna. (%)	type 2 static (%)	type 1 + 2 dyna. (%)
eon	4006	14381689	34.95	52.32	37.09	62.18
mcf	678	27299410	60.18	79.92	64.45	80.37
perl	4920	29600039	86.63	68.92	89.59	70.51
vortex	7830	32691578	91.74	84.14	93.74	85.52
vpr	1917	38016432	81.27	54.71	86.18	66.32
gcc	18166	34023907	79.58	49.28	86.03	53.56

line size is 32B and contains 8 instructions. We impose a limitation that in each cache line, there is at most one extended branch instruction with correlation information, which means that there is at most one extended branch instruction in every 8 instructions. The limitation has minor impact in our scheme. As shown in [5], about 40% of conditional branches have distance greater than 10 instructions. Moreover, only a part of conditional branches will be converted into extended branches. Each entry of EBIT is 8-bit information. Bit 7 indicates whether the cache line has an extended branch instruction. Bit 6 to 4 encodes the position of the extended branch in the cache line. Bit 3 to 0 records the correlation information for the extended branch if there is one. The size of EBIT is 4K bits.

The EBIT is updated with new pre-decoded bits while an I-cache line is refilled after a miss. During each instruction fetch stage, the EBIT is accessed to detect an extended branch instruction and obtain the corresponding correlation information. If the current instruction is an extended branch, further BHT look-up is not necessary. Otherwise, BHT is accessed. Thus, the EBIT access is done before any necessary BHT access. Since the EBIT is small, we assume EBIT access and BHT access together can be done in instruction fetch stage. Note if the current instruction is an original branch, we end up consuming more power since we have to access EBIT too. The validity of correlation-based static prediction relies on the fact that a large percentage of conditional branches can be predicted statically, as shown in Table 1. The power consumption of EBIT is modeled and counted in our experiments.

After we identify all the statically predictable branches and their correlation patterns, we use a compiler (binary-to-binary translator) to convert original branches into extended branches with correlation information properly and get a new transformed program binary. Then, performance results are collected using a modified Simplescalar simulator supporting static branch target buffer and correlation-based static prediction.

4 Experiments and Results

We use Simplescalar 3.0 plus wattch version 1.02 for performance simulation and power analysis. Simplescalar is a cycle-accurate superscalar processor simulator and Wattch is a power analysis model that can be integrated into Simplescalar simulator to track cycle-by-cycle power usage. In Wattch power model, branch predictor power consumption is modeled as three main components, branch direction predictor power, BTB power, and RAS (return address stack) power. BTB and BHT dominate the overall branch predictor power consumption (over 95%). We simulated a typical 8-wide superscalar processor. We choose BTB and BHT configurations comparable to the ones in [5].

We chose six SPEC2000 integer benchmarks that exhibit relatively worse branch prediction performance. They are, eon-cook, mcf, perl, vortex, vpr, gcc. Each benchmark was fast-forwarded 1 billion instructions then simulated for 200 million instructions. The profile information for each benchmark is gathered using standard test inputs. Then standard reference inputs are used to measure performance. Unless explicitly stated, all the branch predictor power consumption results reported are obtained under Wattch non-ideal aggressive clock-gating model (cc3). In this clock-gating model, power is scaled linearly with the number of ports or unit usage. When a unit is inactive, it will dissipate 10% of the maximum power. The power consumption of BTB preloading, phase identification and EBIT is measured using Wattch array structure power model.

Our scheme is built upon the scheme in [5]. We assume only branch instructions will lookup BTB. Thus, BTB updates account for almost half of the BTB activities.

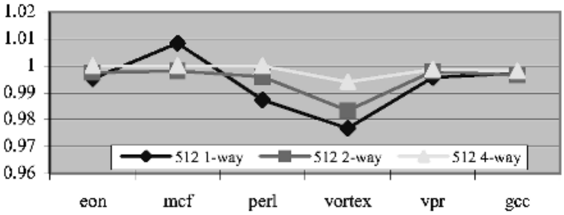


Fig. 4. Normalized Address Hit Rate for Static BTB

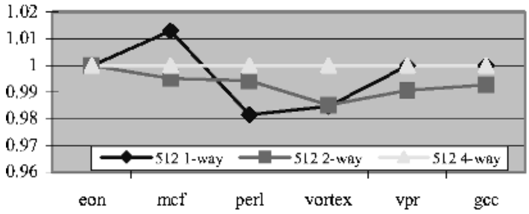


Fig. 5. Normalized IPC for Static BTB

First, we present the results for static branch target buffer. In our study, we assume a 16K PAg direction predictor without static prediction. Three common BTB configurations are examined: 512-set 1-way, 512-set 2-way and 512-set 4-way. Figure 4 shows the normalized address hit rate of static branch target buffer scheme. The address hit rate is normalized to original branch target buffer design. From Figure 4, the impact of our static branch target buffer to address hit rate is minor. For 512-set 1-way configuration, the average degradation in hit rate is 0.66%. For 512-set 2-way configuration, the average degradation is 0.51%. For 512-set 4-way configuration, the average degradation is 0.15%. From the results, we also observe that the degradation on address hit rate becomes even smaller with the increase of the number of entries in

BTB, since now more branches could be preloaded into BTB upon a phase change. *Mcf* benchmark under 512-set 1-way configuration is a corner case in which the address hit rate under static BTB is better than original BTB design. The reason may be destructive aliasing effect. Now that we seldom update BTB, we have much less chance to improperly kick out some branches from BTB.

Figure 5 shows the normalized IPC for static branch target buffer scheme. Note any optimization of power consumption to one component of the processor should not sacrifice the processor performance significantly. Otherwise, the program will run for a longer time and the power savings from the optimized component may be easily killed by more power consumption in other components. For 512-set 1-way BTB configuration, the average IPC degradation of static BTB is 0.34%. For 512-set 2-way configuration, the average IPC degradation is 0.69%. The IPC degradation should become smaller with a larger BTB. 512-set 1-way configuration is better because of the corner case benchmark *mcf*, which achieves better IPC under 512-set 1-way static BTB. The reason is explained earlier. For 512-set 4-way configuration, the degradation is very small thus is not visible in the graph.

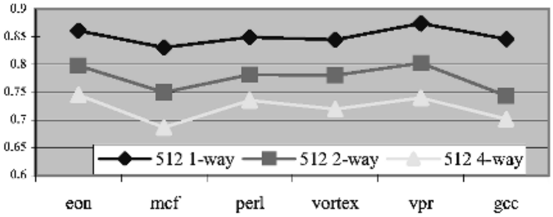


Fig. 6. Normalized Power Consumption for Static BTB

Figure 6 shows the normalized power consumption for the whole branch predictor (BTB + direction predictor) with static BTB design against one with original BTB design. The power saving comes from mostly eliminated BTB updates. The power consumed by BTB preloading and phase tracking reduces the saving but our experiments show they have insignificant impact. After all, our implemented phase tracking hardware is very simple and phase changes are very low-frequency events during execution. For 512-set 1-way configuration, the average power reduction for the branch predictor is 14.89%. For 512-set 2-way configuration, the average power reduction is 22.41%. For 512-set 4-way configuration, the average power reduction is 27.88%. For each configuration, static BTB is able to reduce the BTB activities (dynamic power) by almost half. A larger BTB leads to larger power reduction in percentage because the percentage of BTB power is larger in the whole branch predictor power.

Next, we present the results for correlation-based static prediction. Static prediction works along with a dynamic hardware predictor, which handles branches not predicted by static prediction. A large number of hardware branch prediction schemes have been proposed in the literature. Each one may have its own trade-off in terms of prediction rate, hardware overhead and power consumption. It is impossible for us to examine all of them. In this paper, we limited our scope to two types of predictors, i.e., gshare [9] and PAg [1, 2, 10]. We studied these two types of predictors rather than the default tournament predictor in Alpha 21264 because they are the most basic ones and they can represent two large categories of predictors, i.e. global history based predictors and local history based predictors. We studied them separately to get a deep understanding of the impacts of correlation-based static prediction to different types of predictors. Further, we can derive the impact of our proposed technique to more complicated predictors like the tournament predictor in Alpha, which is basically composed of two global predictors and one local predictor.

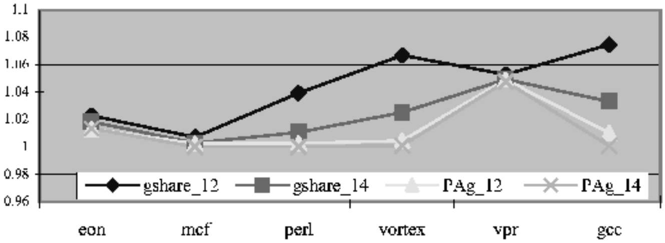


Fig. 7. Normalized Direction Prediction Rate

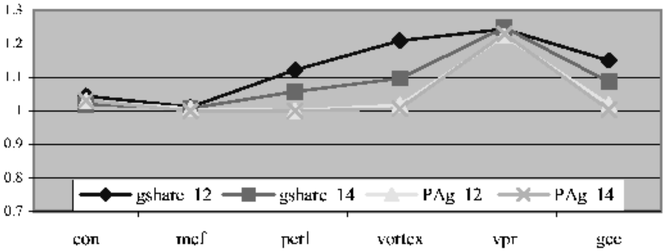


Fig. 8. Normalized IPC

Figure 7 shows branch direction prediction rate under our correlation-based static prediction and dynamic prediction hybrid scheme. The prediction rate is normalized to original pure dynamic direction predictor. In the study of direction predictor, we assume a 512-set 4-way BTB configuration. For gshare-based predictors, we experimented two sizes, 4K and 16K. For PAg based predictors, we also explored two configurations, 4K first level entries –11 bit local history and 16K first level entries – 13 bit local history. From the results, correlation-based static prediction helps prediction rate for gshare-based predictor significantly. For a 4K size gshare predictor, the average improvement in prediction rate is 4.39%. For a 16K size gshare predictor, the average improvement is 2.32%. With the increase of predictor size, the improvement on prediction rate due to correlation-based static prediction becomes smaller. Correlation-based static prediction cannot achieve significant improvement for PAg based predictors, since collision has much smaller impact to the performance of PAg based predictors. For a 4K size PAg predictor, the average improvement is 1.37%. For a 16K size PAg predictor, the average improvement is 1.05%.

Figure 8 shows normalized IPC for the same configurations. Improved direction prediction rate normally leads to improved IPC. For 4K gshare predictor, the average improvement is 13.02%. For 16K gshare predictor, the average improvement is 8.61%. For 4K PAg predictor, the average improvement is 4.97%. For 16K PAg predictor, the average improvement is 4.52%. The improvement on PAg predictors normally is small since the prediction rate improvement is small. Benchmark vpr is a corner case, in which correlation-based static prediction helps improve IPC over 20% in all configurations. Destructive collision effect in *vpr* benchmark must be severe.

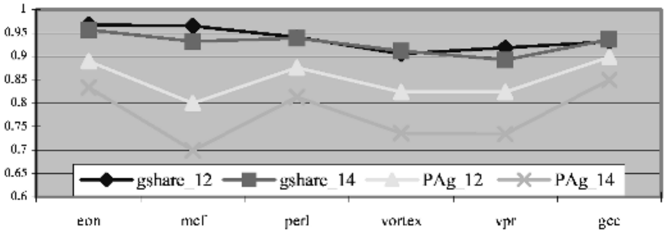


Fig. 9. Normalized Predictor Power Consumption

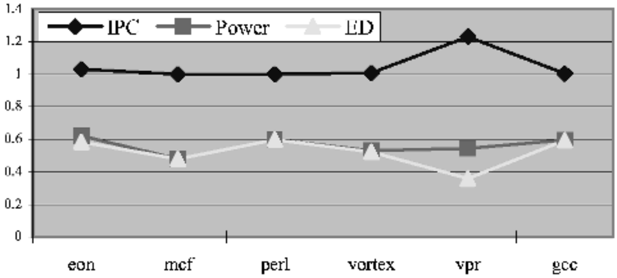


Fig. 10. Results for Combined Architecture

Figure 9 shows the normalized whole branch predictor power consumption under correlation-based static prediction and dynamic prediction hybrid scheme. Our static prediction helps reduce branch predictor power consumption significantly. First, those branches that can be statically predicted will not look up the large BHT, so a large part of BHT lookups are eliminated. Moreover, for a statically predicted branch, only global or local history is updated for the branch, no update is performed to the corresponding 2-bit counter or other finite state machines. For a two-level predictor such as gshare or PAg, this means only the first level of the branch predictor needs state updates. For 4K gshare predictor, the average power reduction is 6.2%. For 16K gshare predictor, the average power reduction is 7.2%. For 4K PAg predictor, the average reduction is 14.8%. For 16K PAg predictor, the average reduction is 22.2%. The reduction for gshare predictors is much smaller since there is no local history table and the power consumed by the EBIT is relatively large.

Our static BTB and correlation-based static prediction could be integrated together into the processor, so that we can achieve significant branch predictor power reduction at the same time with at least no performance degradation in branch predictor. For some benchmarks, the performance of branch predictor is actually improved. Figure 10 shows the results after integration. All the results are normalized against original branch predictor design. We assume a 512-set 4-way BTB configuration and a 16K PAg direction predictor. From the results, for all benchmarks, our optimizations never degrade IPC, i.e., the processor performance. For benchmark *vpr*, the performance is improved significantly instead. The average power consumption reduction is 43.86%. We also show the energy-delay product results, which is a metric to show the trade off between power consumption and performance. The average reduction in energy-delay (ED) product is even better, which is 47.73%.

5 Related Work

In [5], the authors use two major techniques to reduce branch predictor power consumption, banking and PPD (prediction probe detector). PPD is a specialized hardware that records whether lookups to the BTB or the direction-predictor is necessary, therefore saving power by reducing lookups to BTB and BHT when combined with clock gating. Our approach uses two static techniques to reduce the activities of branch predictor further. We propose static BTB to eliminate most of the BTB updates. We propose correlation-based static prediction to eliminate a large part of BHT lookups and updates. Correlation-based static prediction also helps alleviate pressure on the size of BHT, since collisions are reduced significantly. Our results show that we can reduce branch predictor power consumption further upon the optimizations in [5] by around 50% . Hybrid static and dynamic branch prediction has been proposed before for reducing destructive aliasing [6, 22]. Using a profile-based approach similar to ours in this paper, researchers are able to improve branch prediction rate significantly by combining static prediction with some well-known dynamic predictors. However, nobody has proposed using such a hybrid predictor for reducing branch predictor power consumption. In our study, we characterize the power behavior for static and dynamic hybrid branch predictor and claim that such a hybrid predictor is beneficial for both reducing collision and power consumption. We also extend the conventional static prediction by introducing a correlation-based static prediction approach. Smith and Young [3,7,8] have studied branch correlation from a compiler perspective. They have proposed a number of compiler techniques to identify correlated branches using program analysis. Their study is different from ours because their approach is to transform correlated branches in a manner so that they can be correctly predicted under a dynamic hardware predictor rather than directly encoding a correlation pattern into a branch instruction.

6 Conclusion

In this paper, we proposed two optimizations to reduce the power consumption of the branch predictor in a high-performance superscalar processor. We raised the idea of static branch target buffer to eliminate most of the update power of BTB. We also pointed out that a hybrid branch predictor combining static prediction and dynamic prediction not only can reduce branch predictor size and destructive aliasing but branch prediction power consumption as well. We extended the conventional static prediction by putting branch correlation pattern into spare bits in the conditional branches when they are not used. Such correlation-based static prediction can further improve static branch prediction rate comparing to single direction static prediction. We explored the effects of the two proposed optimizations on the trade off between performance and power consumption. Our simulation results show that the integration of the proposed optimizations can reduce the branch predictor power consumption by 44% and branch predictor energy-delay product by 48% while never degrading overall processor performance.

References

[1] T. Y. Yeh, and Y. N. Patt. "Two Level Adaptive Branch prediction". 24th ACM/IEEE International Symposium on Microarchitecture, Nov. 1991.

[2] T. Y. Yeh, and Y. N. Patt. "A Comparison of Dynamic Branch Predictors that Use Two levels of Branch History". 20th Annual International Symposium on Computer Architecture, May 1996.

[3] Cliff Young, Nicolas Gloy, and Michael D. Smith. "A Comparative Analysis of Schemes For Correlated Branch Prediction". ACM SIGARCH Computer Architecture News, Proceedings of the 22nd annual International Symposium on Computer Architecture May 1995, Volume 23 Issue 2.

[4] S. Sechrest, C. C. Lee, and Trevor Mudge. "Correlation and Aliasing in Dynamic Branch Predictors". ACM SIGARCH Computer Architecture News, Proceedings of the 23rd annual international symposium on Computer architecture May 1996, Volume 24 Issue 2.

[5] D. Parikh, K. Skadron, Y. Zhang, M. Barcella, and M. Stan. "Power Issues Related to Branch Prediction". In Proc. of the 2002 International Symposium on High-Performance Computer Architecture, February, 2002, Cambridge, MA.

[6] Harish Patil and Joel Emer. "Combining static and dynamic branch prediction to reduce destructive aliasing". Proceedings of the 6th Intl. Conference on High Performance Computer Architecture, pages 251-262, January 2000.

[7] Cliff Young, Michael D. Smith. "Improving the Accuracy of Static Branch Prediction Using Branch Correlation". ASPLOS 1994: 232-241.

[8] Cliff Young, Michael D. Smith. "Static correlated branch prediction". TOPLAS 21(5): 1028-1075. 1999.

[9] S. McFarling. "Combining branch predictors". Tech. Note TN-36, DEC WRL, June 1993.

[10] Shien-Tai Pan, Kimming So, Joseph T. Rahmeh, “Improving the Accuracy of Dynamic Branch Prediction Using Branch Correlation”, ASPLOS 1992: 76-84.

[11] Michael Haungs, Phil Sallee, Matthew K. Farrens. "Branch Transition Rate: A New Metric for Improved Branch Classification Analysis". HPCA 2000: 241-250.

[12] D. Grunwald, D. Lindsay, and B. Zorn. "Static methods in hybrid branch prediction". In Proc. Of the International Conference on Parallel Architectures and Compilation Techniques (PACT), Oct. 1998. Pages:222 – 229.

[13] A. S. Dhodapkar and J. E. Smith, "Managing Multi-Configuration Hardware via Dynamic Working Set Analysis," Proc. of the 29 Intl. Sym. on Computer Architecture, May 2002, pp. 233 –244.

Choice Predictor for Free

Mongkol Ekpanyapong, Pinar Korkmaz, and Hsien-Hsin S. Lee

School of Electrical and Computer Engineering
Georgia Institute of Technology
Atlanta, Georgia 30332
{pop, korkmazp, leehs}@ece.gatech.edu

Abstract. Reducing energy consumption has become the first priority in designing microprocessors for all market segments including embedded, mobile, and high performance processors. The trend of state-of-the-art branch predictor designs such as a hybrid predictor continues to feature more and larger prediction tables, thereby exacerbating the energy consumption. In this paper, we present two novel profile-guided static prediction techniques— Static Correlation Choice (SCC) prediction and Static Choice (SC) prediction for alleviating the energy consumption without compromising performance. Using our techniques, the hardware choice predictor of a hybrid predictor can be completely eliminated from the processor and replaced with our off-line profiling schemes. Our simulation results show an average 40% power reduction compared to several hybrid predictors. In addition, an average 27% die area can be saved in the branch predictor hardware for other performance features.

1 Introduction

Advances in microelectronics technology and design tools for the past decade enable microprocessor designers to incorporate more complex features to achieve high speed computing. Many architectural techniques have been proposed and implemented to enhance the instruction level parallelism (ILP). However, there are many bottlenecks that obstruct a processor from achieving a high degree of ILP. Branch misprediction disrupting instruction supply poses one of the major ILP limitations. Whenever a branch misprediction occurs in superscalar and/or superpipelined machines, it results in pipeline flushing and refilling and a large number of instructions is discarded, thereby reducing effective ILP dramatically. As a result, microprocessor architects and researchers continue to contrive more complicated branch predictors aiming at reducing branch misprediction rates.

Branch prediction mechanisms can be classified into two categories: static branch prediction and dynamic branch prediction. Static branch prediction techniques [1,6,17] predict branch directions at compile-time. Such prediction schemes, mainly based on instruction types or profiling information, work well for easy-to-predict branches such as while or for-loop branches. Since the static branch prediction completely relies on information available at compile-time, it does not take runtime dynamic branch behavior into account. Conversely,

P.-C. Yew and J. Xue (Eds.): ACSAC 2004, LNCS 3189, pp. 399–413, 2004.

dynamic branch prediction techniques [12,14,16] employ dedicated hardware to track dynamic branch behavior during execution. The hybrid branch predictor [12], one flavor of the dynamic branch predictors, improves the prediction rate by combining the advantages demonstrated by different branch predictors. In the implementation of a hybrid branch predictor, a *choice predictor* is used to determine which branch predictor's results to use for each branch instruction fetched. Introducing a choice predictor, however, results in larger die area and additional power dissipation. Furthermore, updating other branch predictors that are not involved in a prediction draws unnecessary power consumption if the prediction can be done at compile-time. Given the program profiling information, a static choice prediction could be made by identifying the suitable branch predictor for each branch instruction. For example, for a steady branch history pattern such as 000000 or 10101010, the compiler will favor the local branch predictor. On the other hand, for a local branch history pattern of 01011011101 and global branch history pattern of 00**1**1100**1**11000**1**1100**1** (**boldface** numbers correspond to the branch history of this target branch) it will bias toward the global predictor over the local predictor, because the global pattern history shows a repetition of the sequence 001 where 1 corresponds to the target branch.

The organization of this paper is as follows. Section 2 describes related work. Section 3 is devoted to our schemes. Section 4 presents our experimental framework. Results of power, areas and performance are presented in Section 5. Finally the last section concludes this work.

2 Related Work

Most of the branch prediction techniques focus on exploiting the local behavior of each individual branch as well as the global branch correlation to improve prediction accuracy, either at static compile-time or dynamic runtime. Static techniques include two major schemes— profile-guided and program-based schemes. Profile-guided schemes collect branch statistics by executing and profiling the application in advance. The compiler then analyzes the application using these statistics as a guide and regenerates an optimized binary code. Program-based schemes tackle branch prediction problems at source code, assembly, or executable file level without any advanced profiling. One early study on using profile-guided branch prediction was done by Fisher and Freundenberger [6] , in which they showed that profile-guided methods can be very effective for conditional branches as most of the branch paths are highly biased to one direction and this direction almost remains the same across different runs of the program. Ball and Larus [1] later studied a program-based branch prediction method by applying simple heuristics to program analysis at static compilation time for generating static branch predictions.

One important characteristics of branch prediction is that a branch can either exhibit self-correlation or can be correlated with other branches. Yang and Smith [17] proposed a static correlated branch prediction scheme using *path profiling* to find the correlated paths. After identifying all the correlated paths, the technique either duplicates or discriminates the paths depending on the type of correlation. Due to path duplication, their technique increases the code size while reducing misprediction rate.

In spite of the hardware savings, static branch prediction is infeasible for all the branches in a program since a branch can demonstrate very dynamic behavior due to various correlations and will not be strongly biased to one direction or another in their lifetime. Therefore, most of the sophisticated branch prediction mechanisms focus on dynamic prediction mechanisms. Dynamic branch predictors make predictions based on runtime branch direction history. Yeh and Patt [16] introduced the concept of two-level adaptive prediction that maintains a first level N-bit branch history register (BHR) and its corresponding 2^N entry pattern history table (PHT) as a second level for making predictions. The BHR stores the outcomes of the N most recently committed branches used to index into the PHT in which each entry contains a 2-bit saturating up-down counter. They studied both local and global prediction schemes. Local prediction schemes keep the local history of individual branches while global prediction schemes store the global direction history of a number of branches equal to the history register size.

McFarling [12] pioneered the idea of hybrid branch prediction that uses a meta-predictor (or choice predictor) to select a prediction from two different branch predictors. The two branch predictors studied in his paper were bimodal and gshare branch predictors. The bimodal branch predictor consists of a 2-bit counters array indexed by the low order address bits of the program counter (PC). The gshare predictor, which was also christened by McFarling in the same paper is a two-level predictor that exclusive-ORs the global branch history and the branch PC address as the PHT index to reduce destructive aliasing among different branches sharing the same global history pattern. The choice predictor, also a 2-bit counters, is updated to reward the predictor generating correct prediction. Chang et al. [3] studied branch classification. Their classification model groups branches based on profiling data. They also proposed a hybrid branch predictor which takes the advantages of both static and dynamic predictors. Using the profiling data, they perform static prediction for those branches that strongly bias to one direction in their lifetime. Their work is analogous to ours in the sense that we both employ static and dynamic branch prediction method. Comparison and simulation data will be presented and discussed in Section 5. Another work presented by Grunwald et al. in [7] also adopts static prediction for a hybrid predictor. Despite a large experimental data were presented, it remains unclear about their algorithms with respect to how they derive the choice prediction directions at static compile-time. In addition, they compared their static prediction scheme with only McFarling hybrid prediction scheme, while we compare our technique against several other hybrid branch predictors and evaluate the impact to both power and die area. Recently, Huang et al. [4] proposed an energy efficient methodology for branch prediction. Their baseline case is a **2Bc-gskew-pskew** hybrid branch predictor. They used profiling to find out the branch predictor usage of different modules in a program and used clock

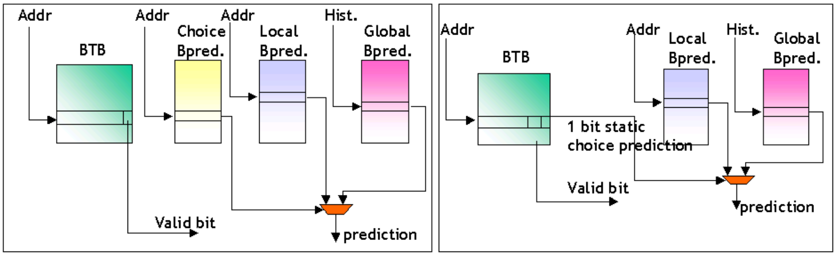


Fig. 1. Branch prediction lookup schemes.

gating to shut down the unused predictors of the above hybrid branch predictor. Different from them, we considered many hybrid branch prediction schemes and we collected profile data for each branch instead of for each module.

3 Static Prediction Generation

Profiling feedback is now a widely accepted technology for code optimization, in particular for static architectures such as Intel/HP's EPIC, we propose a new methodology that utilizes profiling data from prior executions, classifies branches according to the types of correlation exhibited (e.g. local or global), and then decides which prediction result to use. During profile-guided recompilation, these decisions are embedded in the corresponding branch instructions as static choice predictions. For example, the branch hint completer provided in the Itanium ISA [5] can be encoded with such information.

The basic branch prediction lookup scheme for a hybrid branch predictor with a hardware choice predictor and our scheme with static choice prediction are illustrated in Figure 1. In our scheme, the static choice prediction is inserted as an extra bit in the modified branch target buffer (BTB) entry. For each branch predicted, both the local and global predictors are accessed and the prediction implied by the static choice prediction bit in the indexed BTB entry is chosen. The critical path for this branch predictor is not lengthened with such a mechanism, hence no impact to clock speed. Furthermore, using this bit to clock gate the branch predictor might lead to further power reduction, however, it is not explored in this paper.

Most of the hybrid branch predictors with a dynamic choice predictor [9, 12] update all the branch prediction components for each branch access. This is because that, in a dynamic choice predictor, the choice predictor is updated dynamically depending on the prediction results of both branch predictors and for the further accesses to the same branch address there is uncertainty about which branch predictor will be used, hence updating both of them will result in more accuracy. In our model, we update only the branch predictor whose prediction is used, since every branch is already assigned to one of the predictors and updating only the assigned branch predictor is necessary. In our case, updating both branch predictors would not only consume more power but also increase the likelihood of aliasing.

In the following sections, we propose and evaluate two enabling techniques — *Static Correlation Choice (SCC)* prediction and *Static Choice (SC)* prediction from power and performance standpoints.

3.1 SCC Model

In the SCC model, we profile and collect branch history information for each branch. We apply this technique to a hybrid branch predictor that consists of a local bimodal branch predictor [15] and a global two-level branch predictor [16]. The algorithm for the SCC model with the hybrid branch predictor is described in the following steps:

1. If a branch is biased to one direction either *taken* or *not taken* during its lifetime in execution, we favor its prediction made by the bimodal branch predictor. The bias metric is based on a default *threshold* value that represents the execution frequency of the direction of a branch (e.g. 90% in this study, this is based on our intuition that higher than 90% hit rate is acceptable).
2. To model the bimodal branch predictor, we count the total number of consecutive *taken*'s and consecutive *not taken*'s for each branch collected from profile execution. This count based on the local bimodal branch predictor is denoted by C_{LP}. For example, if the branch history of a particular branch is 111100000101010: the number of consecutive ones is 4-1 = 3 and number of consecutive zeros is 4, therefore, C_{LP} = 3+4 = 7.
3. To model the global branch predictor, we collect global history information for each branch on-the-fly during profile execution and compare it against all prior global histories collected for the same branch. If the last k bits of the new global history match the last k bits of any prior global history, then the new prediction is called to be within the same history group. There are 2^k possible groups in total. For each branch that is included in a group, we count the total number of consecutive *taken*'s and consecutive *not taken*'s. At the end of the profile run, we sum up the consecutive counts including *taken* and *not taken* for each history group and denote the value by C_{GP}. For example, assume we have four history groups (k=2) — 00, 01, 10 and 11 for a profile run. For a particular target branch after the profile execution, we have a branch history 101000001111 for the 00 group, 11111111110 for the 01 group, 1110 for the 10 group, and 1000000 for the 11 group. Then the summation for this global branch predictor, for this particular branch would be C_{GP} = 7+9+2+5 = 22. Note that the history does not include the direction of the current reference.
4. C_{LP} and C_{GP} values are collected after the profiling execution. The static choice prediction is made off-line by comparing the values of C_{LP} and C_{GP}. The final choice, provided as a branch hint, as to which predictor to use for each branch is determined by favoring the larger value. In other words, if

C_{LP} is greater than C_{GP}, the choice prediction uses the prediction made by the bimodal predictor otherwise the prediction of the global branch predictor is used.

The SCC model basically targets McFarling's hybrid branch predictor yet collects these information at static compile-time. As aforementioned, McFarling's hybrid branch predictor consists of a bimodal local predictor and a gshare global predictor. The justification behind the calculation of C_{LP} (a metric for bimodal branch prediction) is that, for a bimodal predictor the more the branch result stays in state 00 (strongly not-taken) or 11 (strongly taken), the more stable the prediction will be. On the other hand, C_{GP} of a branch is the metric for the global branch prediction and its calculation is based on counting the number of occurrences of consecutive takens and not-takens (0's and 1's) for this branch for the possible number of different branch histories depending on the length of history. This is similar to the two-bit saturating counters which are chosen by the global history register in the gshare scheme.

3.2 SC Model

In the SC model, static choice predictions completely rely on the results collected from the software-based choice predictor of an architecture simulator. During profiling simulation, we collect the information with respect to how many times the choice predictor is biased to the bimodal predictor versus the global branch predictor for each branch. The final static choice prediction then relies on the majority reported from the profiling simulation.

4 Simulation Framework

Our experimental framework is based on sim-outorder from SimpleScalar toolkit version 3.0 [11]. We modified the simulator to (1) model a variety of hybrid branch predictors , (2) collect the profiling information for the SCC and SC models, and (3) perform static choice branch prediction. Table 1 shows the parameters of our processor model. The SPEC CPU2000 integer benchmark suite [8] was used for our evaluation. All of the benchmark programs were compiled into Alpha AXP binaries with optimization level -O3. All the data presented in Section 5 were obtained through runs of one billion instructions. Since profiling is involved, the experiments were performed among *test*, *train* and *reference* profiling input sets while all the performance evaluation results come from *reference* input set. In other words, we collected different profiling results in order to analyze the impact of our proposed mechanisms with different profiling input sets.

As our proposed technique provides an opportunity to eliminate the choice predictor hardware, we evaluate and quantify the overall power improvement using Wattch [2] toolkit due to the absence of a hardware choice predictor. We modified Wattch to enable clock-gating in different functional blocks of a branch predictor including the BTB, the local, global, and choice predictors, and return address stack.

Table 1. Parameters of the processor model.

Execution Engine	Out-of-order
Fetch Width	8 instruction
Issue Width	8 instruction
ALU Units	4 units
Branch Target Buffer	4-way, 4096 sets
Register Update Unit	128 entries
Cache organization	4-way split I- and D-L1: 64 KB each 2 cycle hit latency 32 bytes line
	4-way L2(unified): 512 KB 16 cycle hit latency 64 bytes line
Memory latency	120 core cycles

5 Experimental Results

This section presents our performance and power analysis. In the first experiment, we study the impact of our static models for choice prediction on performance, including branch prediction rate and speedup. The *train* input set in SPECint2000 benchmarks was used for collecting profile information, while the *reference* input set was used for performance evaluation. Results show that our prediction model performs on par or sometimes better than a hardware choice predictor. It is reported in [10] that energy-delay product is sometimes misleading, hence we report the performance and energy separately.

Figure 2 summarizes the branch prediction miss rates from different branch predictors for SPECint2000 benchmarks. For each benchmark program, experiments are conducted with a variety of branch prediction schemes. Among them are **gshare10**, **gshare11**, **gshare12**, **hybrid_g10**, **hybrid_g10+scc**, **hybrid_g10+sc**, **hybrid_g11+scc**, and **hybrid_g11+sc**. The **gshare10**, same as McFarling's gshare scheme [12], indexes a 1024-entry 2-bit counter array by exclusive-ORing the branch address and its corresponding 10-bit global history. Similarly, **gshare11** and **gshare12** perform the same algorithm by simply extending the sizes of their global history to 11 and 12 bits, thereby increasing their corresponding 2-bit counter arrays to 2048 and 4096 entries, respectively. The predictor, **hybrid_g10** uses a hybrid branch predictor approach similar to McFarling's combining branch predictor [12]. It consists of a bimodal predictor, a two-level predictor, and a choice predictor each of them with a size of 1024x2 bits. The **hybrid_g10+sc** is the same as **hybrid_g10** except replaces the hardware choice predictor with a profiling-based choice prediction mechanism using the SC model described in Section 3. Likewise, **hybrid_g10+scc** uses the SCC model for choice predictions. Predictors **hybrid_g11+scc** and **hybrid_g11+sc**

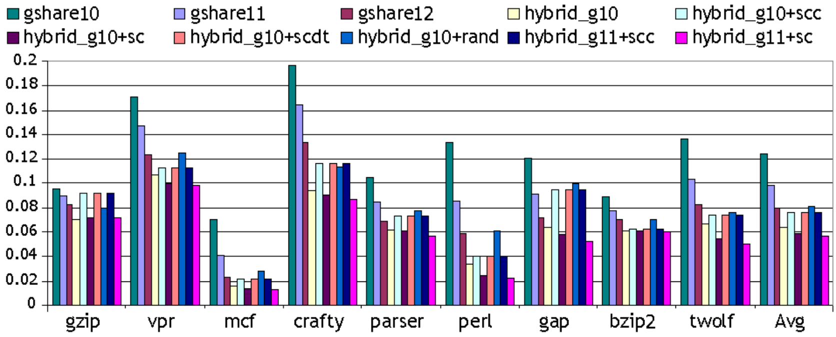


Fig. 2. Miss prediction rates with different branch predictors.

are extended versions of the **hybrid_g10+scc** and **hybrid_g10+sc** models, respectively, as they increase the size of the two-level branch predictor to 2048x2 bits.

Moreover, we also implement the prediction model proposed by Chang et al. [3] which we call SCDT model. In SCDT, profiling is used to classify branches into different groups based on dynamic taken rates and for each group the same branch predictor is used. If the dynamic taken rate of a branch is 0-5% or 95-100% then this branch is predicted using the bimodal predictor, otherwise it is predicted using gshare predictor. If there are a lot of branches that change their behavior dynamically, then SCC captures such behavior better than SCDT. For example, if the behavior of a branch has k consecutive 0's and k consecutive 1's, a bimodal prediction will be better off since it might reduce aliasing in gshare. By contrast SCDT will always use gshare. We also perform experiments using a random choice model which we call RAND model and it randomly selects a branch predictor statically. The **hybrid_g10+scdt** and **hybrid_g10+rand** results are based on the SCDT and RAND models respectively.

As shown in Figure 2, increasing the size of the global branch predictor alone does not perform as well as using a hybrid branch predictor. For example, the **gshare12** predictor consists of more prediction entries than the **hybrid_g10** branch predictor provides (area comparison is shown in Table 2), but none of the benchmarks shows the **gshare12** branch predictor outperforming the **hybrid_g10** branch predictor.

Also shown in Figure 2, instead of having a hardware choice predictor, we can achieve comparable prediction rates using a static off-line choice predictor. Our simulation results show that SCC does not perform as well as SC. This is because the SC model can account for aliasing in its model and hence is more accurate. The difference of these two models is less than 2% in branch miss prediction rates.

Comparing between SCC and SCDT, both schemes provide comparable results. This suggests that branches with varying behavior, as explained earlier,

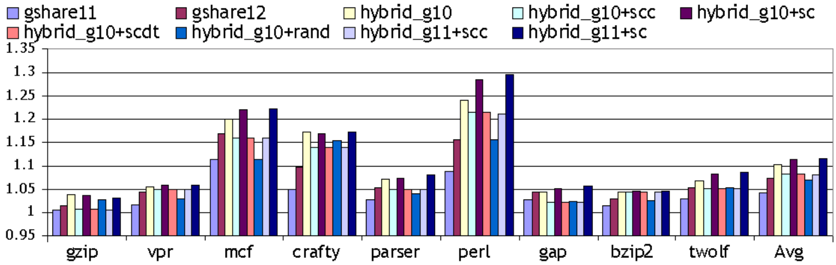


Fig. 3. Normalized speedup with different branch predictors.

rarely occur in SPEC2000. Selecting branch predictors at random does not provide as good an average result as our SCC and SC.

We also show that instead of having a hardware hybrid choice predictor, we can employ a static choice prediction and increase the size of the global branch predictor. The **hybrid_g11+sc** model demonstrates the best prediction rate among others for most of the benchmarks.

Figure 3 shows the normalized performance speedups of various prediction schemes; the baseline in this figure is **gshare10**. The results show that the speedup's improve as the prediction rates increase. We expect the increase will be more significant with deeper, and wider machine.

Previously, we explained that the motivation of our work is to reduce the area and power consumption of a branch predictor while retaining the performance. To this end, we use Wattch to collect the power statistics of the branch predictor and other functional units of the processor. Both dynamic and static power consumption were considered in our evaluation. For each functional block (such as BTB, branch predictor, i-cache, and d-cache), the switching power consumed per access is calculated along with the total number of accesses to that block. Additionally, when a block is not in use, we assume an amount of static power equal to 10% of its switching power is consumed. Note that this amount will increase significantly when migrating to the future process technology. Thus, the elimination of the choice predictor will gain more advantage in overall power dissipation. We also want to mention that we examined the effect of our branch prediction schemes on the power consumption of the branch direction predictor, and we claim improvements on the power consumption of the branch direction predictor.

Figure 4 shows the normalized power consumption values for different branch predictors, relative to the power consumption of **gshare10**. From this Figure and Figure 3, we can tell that for nearly all the benchmarks, **hybrid_g10+sc** yields the best processor performance for little branch prediction power. We can use Figures 3 and 4 as guides in a design space exploration framework, where the power budget of the branch predictor is limited, and a specific performance constraint has to be satisfied. For example, the results in Figure 4 show that the removal of the choice predictor in **hybrid_g10** can reduce the power consump-

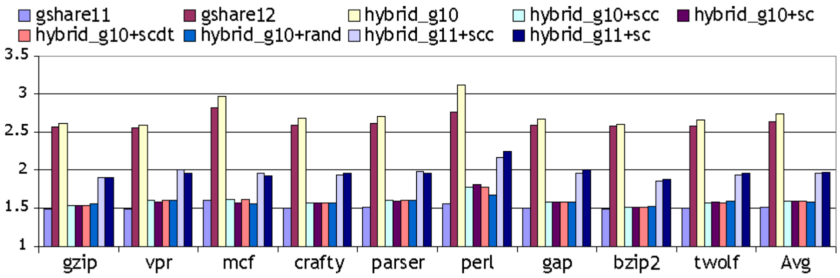


Fig. 4. Normalized power consumption of different branch predictors.

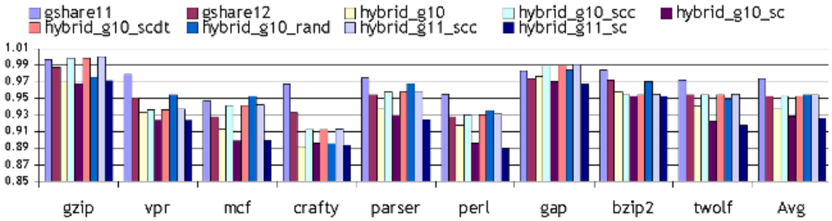


Fig. 5. Normalized processor energy with different branch predictors.

tion to a level comparable to that of **gshare11**. Similarly Figure 3 shows that **hybrid_g10+sc** outperforms **gshare11**, for all the benchmarks. Hence we can deduce that using **hybrid_g10+sc** is more advantageous in terms of both the power dissipation and performance. We present the total energy consumption of the processor in Figure 5. Despite the fact that **gshare10** has lowest power consumption, all other branch predictors outperform **gshare10** in terms of total energy consumption. When we compare the power consumption with static and dynamic methods for the same type of branch predictor, static choice predictor consumes less power. However the total energy consumption depends not only the power consumption but also execution time. Hence **hybrid_g10** model which has better the performance on average than **hybrid_g10+scc** has smaller energy consumption. **Hybrid_g10+sc** instead has the smaller energy consumption than most of branch predictors including **hybrid_g10** since it is faster and consumes less power. Moreover **hybrid_g11+sc** which has higher branch prediction's power dissipation than **hybrid_g10+sc** outperforms all branch predictors in terms of total energy consumption.

Next, we study the impact of profiling on the training input set of our SC and SCC training. We aim to show how our models SCC and SC are affected as a result of various training data. We use three different input sets for profiling: *test*, *train*, and *reference*. The results show little impact on the branch prediction outcomes. The results are detailed in Figure 6 where the baseline is again **gshare10**. Figure 6 shows that SCC is less sensitive to profile information than SC. This is because SC incorporates aliasing information in its model. Let us

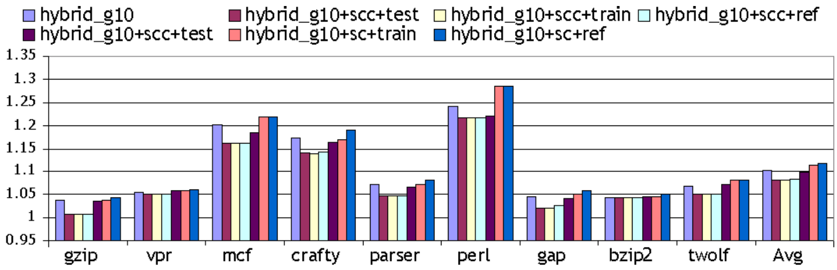


Fig. 6. Normalized speedup on different profiling input sets.

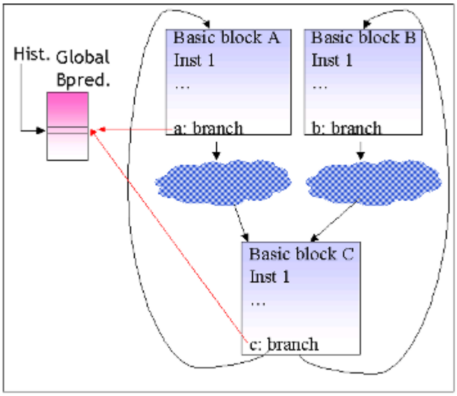


Fig. 7. CFG example showing aliasing impact.

consider the Control Flow Graph (CFG), which is shown in Figure 7. Assume that branches *a* and *c* point to the same location in global branch predictor and also are predicted accurately by a global branch predictor if there is no destructive aliasing. If branches *a* and *c* destructively interfere with each other, this results in profiling say that loop *A*-*C* is called more frequently than loop *B*-*C* hence static choice predictor will assign both branches *a* and *c* to local branch predictor. However on the running input set, if loop *C*-*A* runs more often than loop *B*-*A* then assigning both *a* and *c* to local branch predictor can reduce branch prediction accuracy. Figure 6 also shows that if profile information has the same behavior as the real input set, static choice predictor can outperform hardware choice predictor in most benchmarks.

We then perform experiments using different hybrid branch predictors to show that SC and SCC are equally compatible with different kinds of hybrid branch predictors. In this set of experiments, **gshare10** is our chosen baseline. The results are shown in Figure 8. Note that **hybrid_PAg** is a hybrid branch predictor similar to the one used in Alpha 21264 processor. It consists of a two-level local predictor with a local history table size of 1024x10 bits, local predictor

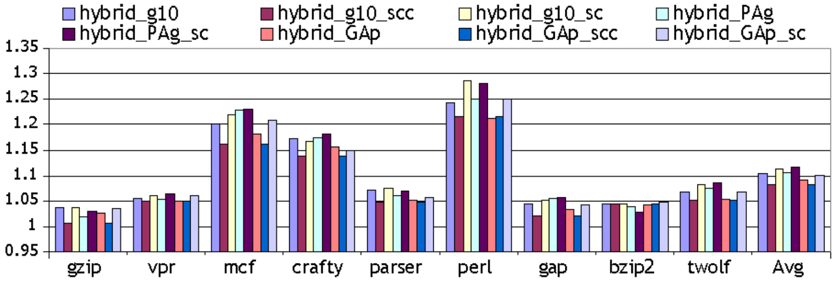


Fig. 8. Normalized speedup on different hybrid branch predictors.

size of 1024x2 bit and with global and choice predictors of size 1024x2 bit. **hybrid_GAp** stands for a hybrid branch predictor with a 1024x2 bit bimodal predictor and four of 1024x2 bit counters instead of one such counter as in **hybrid_g10**.

Since SCC is not intended to target **hybrid_PAg**, i.e it cannot exploit full advantages from local branch predictor in **hybrid_PAg**, we exclude the result of the SCC on **hybrid_PAg**. For example, if we have local history pattern of 1010101010, C_{LP} is 0 and SC will not choose local branch predictor but local predictor in **hybrid_PAg** can predict this pattern accurately.

Results shown in Figure 8 also indicate that SC works well with **hybrid_PAg**.

We now report the power consumption of different branch predictors and total energy consumption of the processor using these branch predictors. Figure 9 shows the normalized power consumption for different hybrid predictors relative to **gshare10**. In this figure, we observe that for **hybrid_g10**, and **hybrid_GAp**, using SC and SCC methods bring an improvement of 42% on average. The average improvement for **hybrid_PAg** is around 37%. The power consumption in **hybrid_GAp** is not too high compared with **hybrid_g10** since clock gating is applied to unused predictors. Figure 10 shows the total energy consumption of the processor using these hybrid predictors. Using SC method with **hybrid_PAg** branch predictor gives the best result in terms of the energy consumption of the processor and this is due to the high speedup obtained using **hybrid_PAg_sc** which is observed on Figure 8.

These results allow the possibility of replacing the hardware choice predictor with our schemes, and reclaim in its corresponding die area. Assuming a static memory array area model, as described in [13], for the branch predictor, the area can be quantified as followings:

$$area_{static\_memory} = 0.6\ (size_w + 6)\ (line_b + 6)\ rbe \qquad (1)$$

where $size_w$ is the number of words, $line_b$ is the number of bits and rbe is an area unit of a register cell. The two +6 terms approximate the overhead for the decoder logic and sense amplifiers. Based on equation 1, we derived the normalized areas of different branch predictors relative to **gshare10** in Table 2. Note that the branch predictor area saved by using our profile-guided SCC and SC

Table 2. Normalized area of hybrid branch predictors.

Branch predictor	Normalized area
gshare11	1.9822
gshare12	4.806
hybrid_g10	2.973
hybrid_g10+scc/sc	1.986
hybrid_g11	3.955
hybrid_g11+scc/sc	2.968
hybrid_PAg	4.946
hybrid_PAg+sc	3.959
hybrid_GAp	3.713
hybrid_GAp+sc/scc	2.726

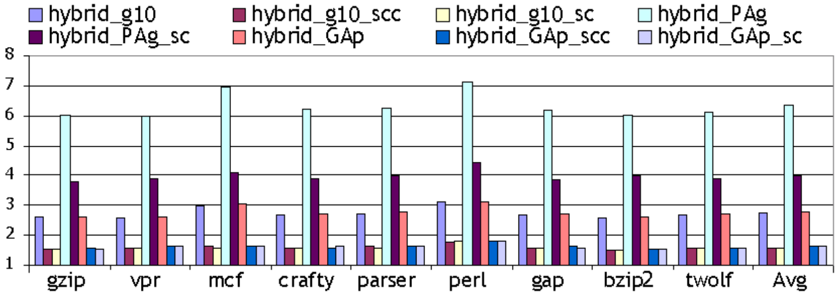


Fig. 9. Normalized power consumption of different hybrid branch predictors.

schemes for **hybrid_g10** predictor is 33.18%. The saving is less for other predictors because these predictors are comprised of more complicated local and global predictors which consume a lot of area. One interesting result in the table shows that the area of **hybrid_GAp+sc/scc** is smaller than the area of **hybrid_g10**. This is due to the fact that fewer decoders are needed for **hybrid_GAp+sc/scc** compared to **hybrid_g10**. The four 1024x2 bit tables in **hybrid_GAp** share the same decoder, hence we need only one 10x1024 decoder and one 2x4 decoder for **hybrid_GAp**, while **hybrid_g10** needs three separate 10x1024 decoders (one for each predictor).

6 Conclusions

In this paper, we study two profile-guided techniques: *Static Correlation Choice* and *Static Choice*, for performing off-line static choice predictions. Our work offers the possibility of eliminating the hardware choice predictor while achieving comparable performance results. In other words, the branch prediction rates attained by dynamic choice predictors can also be achieved using the two proposed

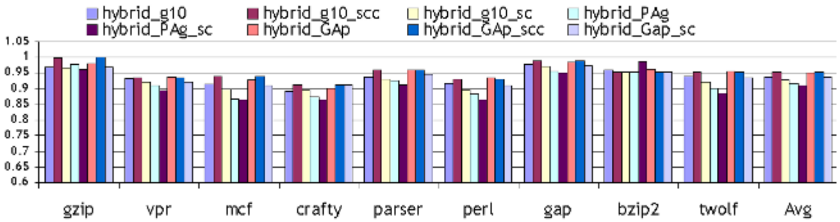


Fig. 10. Normalized processor energy with different hybrid branch predictors.

models, thus resulting in similar performance. The studies we carried out using different input data further indicate that the SC and SCC techniques are largely insensitive to profiling data. By using our techniques, we can reduct the power dissipation of the branch predictor by 40% on average. Moreover, an average saving of 27% in branch predictor area can be saved.

References

1. T. Ball and J. R. Larus. Branch Prediction for Free. In *PLDI-6*, 1993.
2. D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A Framework for Architectural-Level Power Analysis and Optimizations. *ISCA-27*, June 2000.
3. P.-Y. Chang, E. Hao, T.-Y. Yeh, and Y. N. Patt. Branch Classification: a New Mechanism for Improving Branch Predictor Performance. *International Journal of Parallel Programming*, Vol. 24, No. 2:133–158, 1999.
4. Daniel Chaver, Luis Pinuel, Manuel Prieto, Francisco Tirado, and Michael C. Huang. Branch Prediction on Demand: an Energy-Efficient Solution . In *Proceedings of the 2003 International Symposium on Low Power Electronics and Design*, 2003.
5. Intel Corporation. IA-64 Application Developer's Architecture Guide. Intel Literature Centers, 1999.
6. J. A. Fisher and S. M. Freudenberger. Predicting Conditional Branch Directions From Previous Runs of a Program. In *ASPLOS-5*, pages 85–95, 1992.
7. D. Grunwald, D. Lindsay, and B. Zorn. Static Methods in Hybrid Branch Prediction. In *PACT'98*, 1998.
8. John L. Henning. SPEC CPU2000: Measuring CPU Performance in the New Millennium. *IEEE Micro*, July 2000.
9. R. E. Kessler. The ALPHA 21264 Microprocessor. *IEEE Micro*, March/April 1999.
10. H.-H. S. Lee, J. B. Fryman, A. U. Diril, and Y. S. Dhillon. The Elusive Metric for Low-Power Architecture Research. In *Workshop on Complexity-Effective Design*, 2003.
11. SimpleScalar LLC. SimpleScalar Toolkit version 3.0.
http://www.simplescalar.com.
12. S. McFarling. Combining Branch Predictors. Technical Report TN-36, Compaq Western Research Lab, 1993.
13. J. M. Mulder, N. T. Quach, and M. J. Flynn. An Area Model for On-Chip Memories and its Application. *IEEE JSSC*, Vol. 26 No. 2, February 1991.

14. Shien-Tai Pan, Kimming So, and Joseph T. Rahmeh. Improving the Accuracy of Dynamic Branch Prediction Using Branch Correlation. *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, 1992.
15. J. E. Smith. A Study of Branch Prediction Strategies. In *ISCA-8*, 1981.
16. T.-Y. Yeh and Y. N. Patt. Two-Level Adaptive Training Branch Prediction. In *MICRO-24*, 1991.
17. C. Young and M. D. Smith. Static Correlated Branch Prediction. *ACM TOPLAS*, 1999.

Performance Impact of Different Data Value Predictors

Yong Xiao, Kun Deng, and Xingming Zhou

National Laboratory for Parallel & Distributed Processing
Changsha, P.R. China, 410073
yxiao1977@hotmail.com

Abstract. Data value prediction has been widely accepted as an effective mechanism to exceed the dataflow limit in processor parallelism race. Several works have reported promising performance potential. However, there is hardly enough information that is presented in a clear way about performance comparison of these prediction mechanisms. This paper investigates the performance impact of four previously proposed value predictors, namely last value predictor, stride value predictor, two-level value predictor and hybrid (stride+two-level) predictor. The impact of misprediction penalty, which has been frequently ignored, is discussed in detail. Several other implementation issues, including instruction window size, issue width and branch predictor are also addressed and simulated. Simulation results indicate that data value predictors act differently under different configurations. In some cases, simpler schemes may be more beneficial than complicated ones. In some particular cases, value prediction may have negative impact on performance.

1 Introduction

The inevitably increasing density of transistors on one silicon die allows chip designers to put more and more execution resources into single chip to improve performance. However, the presence of data dependences in programs greatly impairs their effort. Value prediction is a speculative technique that uses the previous results of a static instruction to predict the value of the instruction's next output value. Recent studies have shown bright future of using data value prediction to exceed the dataflow limit [2, 3, 6, 10, 12, 13].

Previous studies have introduced many data value predictors. Performance impact of each predictor has been thoroughly investigated respectively. Yet no works have reported the performance comparison of these predictors in a clear way. Such study is helpful in better understanding the effects of value prediction mechanisms and is of great importance when designing an appropriate value predictor. In this paper, we integrate value prediction mechanism in a pipeline implementation and investigate the performance impact of different value predictors. Four previously proposed value predictors, namely last value predictor [1, 2, 3, 4], stride value predictor, two-level value predictor and hybrid (stride+two-level) predictor [4] are used as candidates. Different implementation configurations are also discussed.

P.-C. Yew and J. Xue (Eds.): ACSAC 2004, LNCS 3189, pp. 414–425, 2004.

Simulation analysis indicates that we may need to re-evaluate the impact of data value prediction on performance. With different misprediction penalties and implementation issues, these predictors showed different characteristics. Some predictors may even cause performance degradation. To our surprise, complicated predictors may obtain lower IPC (instructions per cycle) than simpler schemes in some experiments. At last, many factors of value prediction implementation are still ambiguous by now. Further research must be done to fully evaluate data value prediction.

The remainder of the paper is organized as follows: Section 2 discusses recent relevant work in value prediction. Section 3 introduces the microarchitecture implementation. Section 4 summarizes our experimental methodology and presents performance results. Section 5 presents a summary of this work, and proposes some suggestions for future research.

2 Related Works

Value prediction is possible because of value locality - the property of recent values to recur in computer system storage locations [1]. [1] indicates that a large fraction of values computed by the same static load instruction are a repetition set within the 16 most recent values produced by the instruction. Follow-up studies [2, 3] extend value prediction to all register writing instructions. Rychlik et al [5] present a value-predicting wide superscalar machine with a speculative execution core. Their work indicates that many predictions may be useless in enhancing performance.

Wang Kai et al investigate a variety of techniques to carry out highly accurate value prediction [4]. Their simulation analysis shows that the four predictors used in our paper have different static prediction rate and prediction accuracy. Yet the simulation is made without pipeline involved. The limits of performance potential of data value speculation to boost ILP are addressed in [6]. The impact on stride value predictor of some features, such as instruction window size and branch prediction accuracy are discussed and simulated within an idealized experiment environment.

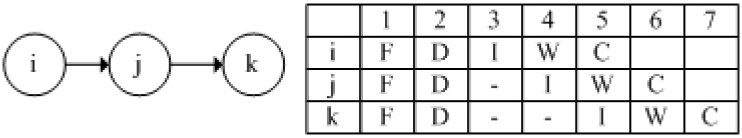
However, there exists no formalized method that defines the effects of value speculation on microarchitecture. Y. Sazeides thus proposes a methodical model for dynamically scheduled microarchitectures with value speculation [7]. The model isolates the parts of a microarchitecture that may be influenced by value speculation in terms of various variables and latency events. Simulation results obtained with a context-based predictor show that value speculation has non-uniform sensitivity to changes in the latency of such events.

3 Microarchitecture

This section first describes a base-microarchitecture and then introduces the value speculation microarchitecture used in the paper. Different data value predictors and misprediction penalty considerations are also introduced respectively.

3.1 Base Microarchitecture

For base microarchitecture we consider an out-of-order superscalar processor model used in SimpleScalar2.0 [15], where the Register Update Unit (RUU) structure unifies issue resources and retirement resources. Memory instructions consist of two operations: address generation and memory access. Fast forward is implemented, thus data dependent instructions can get executed in the same cycle when an instruction writes back its result to physical registers. The pipeline is six-stage implemented: Fetch (F), Dispatch (D), Issue (I), Execution (E), Write back (W) and Commit (C). The pipelined execution of three data dependent instructions *i*, *j* and *k* is shown in Figure 1. One-cycle execution latency is assumed for all instructions, which means the results are available just after I-stage, so E-stage is not shown.



	1	2	3	4	5	6	7
i	F	D	I	W	C		
j	F	D	-	I	W	C	
k	F	D	-	-	I	W	C

Fig. 1. Execution Example of Base Architecture

3.2 Data Value Predictors

Four previously proposed predictors, namely last value predictor (LVP), stride value predictor (SVP), 2-level predictor (2LP), stride+2level hybrid predictor (S2P) are used here as candidates.

Last value prediction predicts the result of an instruction based on its most recently generated value. While stride value prediction predicts the value by adding the most recent value to the stride, namely the difference of the two most recently produced results.

The 2-level value predictor predicts the value based on the repeated pattern of the last several values observed. It consists of two tables: a value history table (VHT) and a pattern history table (PHT). The VHT stores up to four distinct values 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.

The hybrid value predictor is a combination of SVP and 2LP. Compared to the VHT in 2LP, the VHT of the hybrid predictor has two additional fields: state and stride, for the stride prediction. 2LP 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.

The work done by Sang [16], where schemes in [1, 4] are realized without pipeline involved, is referenced here for comparison. Configurations for these predictors used in the paper are summarized in Table 1. A maximum capacity of about 128K (1K=1024) bytes is considered for each predictor. All predictors are directly mapped.

Table 1. Configuration of Predictors

Predictor	Threshold	Saturate At	Update Method	Entry
LVP	2	3	Correct:+1, else:-1	16K
SVP	steady	state=steady	state transition diagram	8K
2LP	3	12	Correct:+3, else:-1	4K
S2P	SVP: steady	steady	state transition diagram	4K
	2LP: 6	12	Correct:+3, else:-1	

3.3 Microarchitecture with Value Prediction

The base machine model is modified to support result value prediction. In addition to adding the value predictor, each RUU is expanded to hold the predicted value and other information for speculative execution.

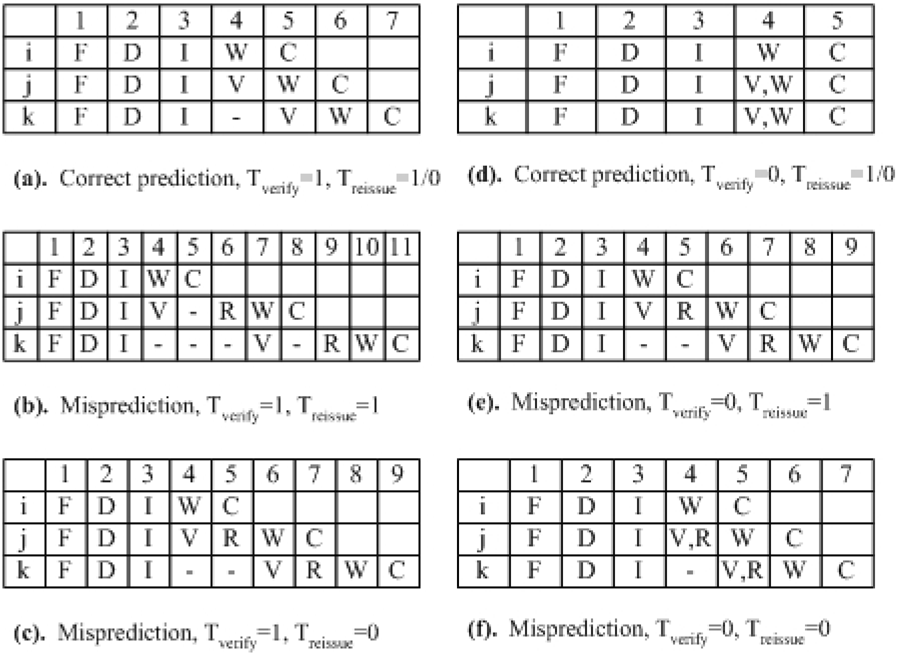
At fetch stage, instruction address is used to access the value predictor. And in our experiments, the number of value predictions made in a cycle is unlimited. Once a prediction is made, the predicted value will be available at dispatch stage. Later dependent instructions will be able to get issued with the predicted value. The data value predictor is updated when an instruction gets committed, namely the predictor can not be updated speculatively. Speculative update will be studied in the future.

Instructions can be speculative or non-speculative, predicted or un-predicted. Speculative instructions have at least one predicted source operand when they get executed; predicted instructions have a predicted result. In our implementation, data value prediction is performed only for load and simple ALU instructions. Branch instructions, store instructions, nops and double-precision instructions are not considered for data value prediction. For the complexity brought by speculative memory access, only simple ALU instructions and address generations can execute speculatively. Moreover, for configuration Conf4 (will be mentioned in section 4.2), when none-zero misprediction penalty is used, address generation is not permitted to execute speculatively. Even if address generation is allowed to do speculative execution, memory access can only be issued when all its operands are final.

A speculatively issued instruction is prevented from completing architecturally and is forced to retain possession of its issue resources until its inputs are no longer speculative. After execution, the speculative results are not broadcast on the result bus. Instead, they are locally written back to the same RUU. Once a prediction is verified, all dependent instructions can either release their issue resources and proceed into the commit stage (in the case of a correct prediction) or restart execution with the correct register values (if the prediction was incorrect). In our work, a predicted instruction can only verify its direct successors. The verified instructions can do the activation successively.

3.4 Misprediction Penalty Considerations

For misprediction penalty, two time parameters - T_{verify} and $T_{reissue}$ are considered. T_{verify} stands for the latency between a predicted instruction finishes execution and dependent speculatively executed instructions get verified. $T_{reissue}$ stands for the latency after a speculatively executed instruction using an incorrectly predicted value gets verified before it can reissue. T_{verify} and $T_{reissue}$ can be either 0 or 1 in our work. Latency that is bigger than 1 will be too pessimistic for our processor model. Figure 2 illustrates the pipelined execution of the three instructions used in figure 1 under all four combinations of T_{verify} and $T_{reissue.}$ R and V stand for reissue and verify respectively. For each combination, two scenarios are considered: (1) both the outputs of *i* and *j* are correctly predicted, (2) both the outputs of *i* and *j* are mispredicted.



	1	2	3	4	5	6	7
i	F	D	I	W	C		
j	F	D	I	V	W	C	
k	F	D	I	-	V	W	C

(a). Correct prediction, T_{verify}=1, $T_{reissue}$=1/0

	1	2	3	4	5	6	7	8	9	10	11
i	F	D	I	W	C						
j	F	D	I	V	-	R	W	C			
k	F	D	I	-	-	-	V	-	R	W	C

(b). Misprediction, T_{verify}=1, $T_{reissue}$=1

	1	2	3	4	5	6	7	8	9
i	F	D	I	W	C				
j	F	D	I	V	R	W	C		
k	F	D	I	-	-	V	R	W	C

(c). Misprediction, T_{verify}=1, $T_{reissue}$=0

	1	2	3	4	5
i	F	D	I	W	C
j	F	D	I	V,W	C
k	F	D	I	V,W	C

(d). Correct prediction, T_{verify}=0, $T_{reissue}$=1/0

	1	2	3	4	5	6	7	8	9
i	F	D	I	W	C				
j	F	D	I	V	R	W	C		
k	F	D	I	-	-	V	R	W	C

(e). Misprediction, T_{verify}=0, $T_{reissue}$=1

	1	2	3	4	5	6	7
i	F	D	I	W	C		
j	F	D	I	V,R	W	C	
k	F	D	I	-	V,R	W	C

(f). Misprediction, T_{verify}=0, $T_{reissue}$=0

Fig. 2. Execution Example with Different T_{verify} and $T_{reissue}$ Configurations

Figure 2 indicates that when T_{verify} equals to 1, value prediction will not improve performance and sometimes will cause performance degradation. Under such situations, value speculation is not beneficial. In our work, only $T_{verify} = 0$ is analyzed.

There can also be structural hazard penalties due to value speculation implementation, especially when the execution resources are limited. Such structural hazard is also well simulated in the experiments.

4 Performance Evaluations

In this section we will introduce our experiment environments and present results obtained from the simulation experiments.

4.1 Benchmarks

The precompiled SPEC CINT95 benchmark suite supplied by SimpleScalar [15] is used in our simulation. Table 2 shows the benchmarks and the specific input sets used. The last column gives the percentage of dynamic instructions that are eligible for data value prediction. All benchmarks were run to completion.

Table 2. Benchmarks and Percentage of Instructions Eligible for Value Prediction

Benchmark	Input Set	VP-Eligible Instructions
099.go	9 9 go-null.in	77.69%
124.m88ksim	m88ksim-ctl.raw	66.38%
126.gcc(cc1)	-O cc1-cccp.i –o cc1-cccp.s	65.78%
129.compress	compress-test.in	35.23%
130.li	li-test.lsp	58.78%
132.ijpeg	-image_file ijpeg-specmun.ppm	74.48%
134.perl	jumble.pl < jumble.in	62.53%
147.vortex	vortex.raw	61.65%

4.2 Architecture Parameters

Our implementation has a 16K bytes directly mapped instruction cache. The first level data cache is 4-way associative, 16K bytes for Con4, 64K bytes for other configurations. Both caches have block sizes of 32 bytes and hit latency of 1 cycle. There is a unified second level 256K bytes 4-way associative cache with 64 byte blocks and a 6-cycle cache hit latency. The first 8 bytes of a data access request to memory will need 18 cycles and the rest of the access will take 2 cycles for each 8 bytes. There is a 16 entry 4-way associative instruction TLB and a 32 entry 4-way associative data TLB, each with a 30 cycle miss penalty. The branch target buffer (BTB) has 512 entries and is 4–way associatively organized. Moreover, an 8-entry return address stack is used.

The latencies of functional units are set as below: Integer ALU 1 cycle, Integer MULT 3 cycles, Integer DIV 20 cycles, FP Adder 2 cycles, FP MULT 4 cycles and FP DIV 12 cycles. All functional units, except the divide units, are pipelined to allow a new instruction to initiate execution each cycle.

Other architecture parameters are summarized in Table 3. Four configurations will be simulated. Conf4, which is very conservative, only allows small window size and issue width. While the next three configurations – Conf8, Conf8_bP and Conf8_bPvP are much closer to those of modern processors. Here "bP" means perfect branch prediction and "vP" means perfect value prediction. With perfect branch prediction, the fetch engine will always get instructions from the correct path and no instructions from the wrong path will be used. With perfect value prediction, value prediction will be used only if the predicted value equals to the final result. Meanwhile the value predictor will be updated immediately with the correct results along with the predictor lookup process, and needs not wait until the instruction gets committed.

Table 3. Processor Configurations

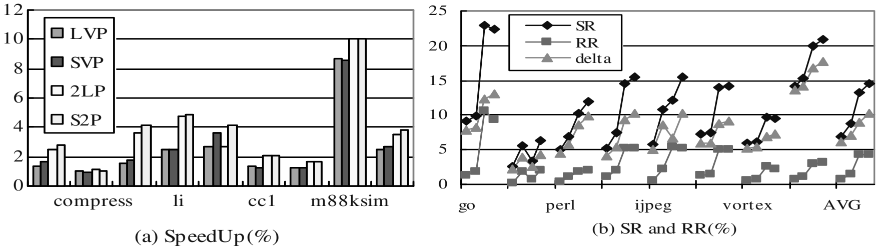
parameter	Conf4	Conf8	Conf8_bP(vP)
instruction window (entries)	16	128	256
load-store queue (entries)	8	128	256
fetch/decode/issue/commit width (instructions/cycle)	4	8	8
functional units (numbers) INT ALU INT MULT/DIV FP Adder FP MULT/DIV Memory Port	 4 1 4 1 2	 8 2 4 2 2	 8 2 4 2 4
branch predictor	bimodal, 2K entries, 3 cycles miss penalty		perfect

4.3 Experimental Results

Two kinds of configurations: (1) T_{verify}=0, $T_{reissue}$ = 0, (2) T_{verify}=0, $T_{reissue}$ = 1 are simulated. To better understand the value speculation performance, we define two metrics: speculation rate (SR) and reissue rate (RR). Speculation rate is defined as the percentage of speculatively executed instructions over all executed instructions including those that are on the wrong path. Reissue rate is the percentage of speculative instructions that get reissued at least one time over all executed instructions.

4.3.1 Prediction Performance Using Conf4

Figure 3 shows the simulation results when $T_{reissue}$ = 0. Figure 4 shows the performance when $T_{reissue}$ = 1. In figure 3(b) and figure 4(b), for each benchmark, 4 groups of results are given; from left to right, these correspond to predictor LVP, SVP, 2LP and S2P respectively. Each group of results consists of 3 parts. The uppermost part (SR) indicates the speculation rate. The next part (RR) indicates the reissue rate, and the next part (delta) indicates the difference of speculation rate and reissue rate, namely the percentage of speculative instructions that are executed correctly over all executed instructions including those that are on the wrong path.



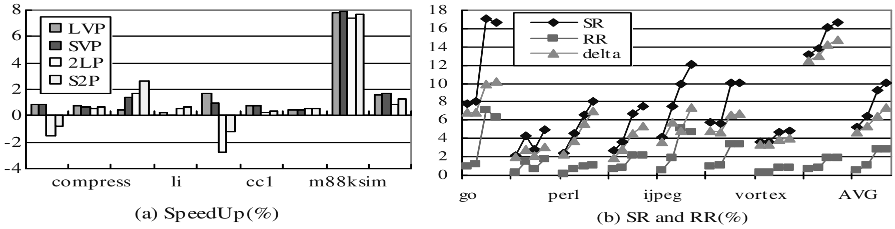
(a) SpeedUp(%)

(b) SR and RR(%)

Fig. 3. Prediction Performance of Four Predictors Using Conf4, T_{verify}=0, $T_{reissue}$=0

Figure 3(a) indicates that S2P performs the best among all predictors, while LVP performs the worst. When $T_{reissue}$ = 0, there is no misprediction penalty except the penalty brought by structural hazards. So the more correct speculations/predictions (delta in figure 3(b)) a scheme performs the more benefits it can obtain. Reissue rate or incorrect prediction rate is not a decisive factor in such a situation.

Figure 4(b) shows similar trend as figure 3(b), except that both the value of SR and RR are decreased a little. When $T_{reissue}$ equals to 1, address generation operations are not allowed to execute speculatively, which is mainly responsible for the difference. The impact of such factor needs further research.



(a) SpeedUp(%)

(b) SR and RR(%)

Fig. 4. Prediction Performance of Four Predictors Using Conf4, T_{verify}=0, $T_{reissue}$=1

Figure 4(a) indicates that SVP obtains the highest average speedup among all four predictors. This is mainly due to the fact that when misprediction penalty is not negligible, prediction accuracy is more crucial than the amount. Although S2P still performs the most correct speculations, the higher reissue rate (figure 4(b)) impairs its benefit. A more noticeable fact is that value speculation may sometimes cause performance degradation. For benchmark *go* and *ijpeg*, negative impact on the performance is performed by both 2LP and S2P. This can be explained by the high reissue rate showed in figure 4(b). Especially for *ijpeg* by 2LP where RR is higher than delta, it's not surprising to obtain the worst performance.

4.3.2 Prediction Performance Using Conf8

In this subsection, Conf8 is used in our experiments. Figure 5 and Figure 6 show the simulation results for $T_{reissue}$ = 0 and $T_{reissue}$ = 1 respectively. As more instructions are allowed to be fetched and issued in one cycle, more instructions get executed speculatively. Meanwhile more wrong predictions are made. As a consequence, both SR and RR are higher in figure 5 and 6 than those in figure 3 and 4.

From figure 5(a), we find again that S2P performs the best among all predictors, while LVP performs the worst. Two other facts are worth mentioning here. First is that the speedup obtained with Conf8 is lower than that with Conf4. One reason is that with larger window size and issue width, the scheduling hardware has more opportunities rearranging instructions. Thus the effect of value prediction is weakened. The relative higher reissue rate is also responsible for performance decline. The other noticeable fact is the negative impact on benchmark *perl* by LVP and SVP. It is because that the number of instructions on the wrong path is greatly increased under LVP and SVP, which in turn result in performance degradation.

Due to the lowest reissue rate and stable behavior among all benchmark programs, LVP performs the best in figure 6(a). Meanwhile, SVP, for its bad performance with *compress* and *perl*, takes the second place. For the relative higher reissue rate, more programs perform worse with value speculation than in figure 4(a).

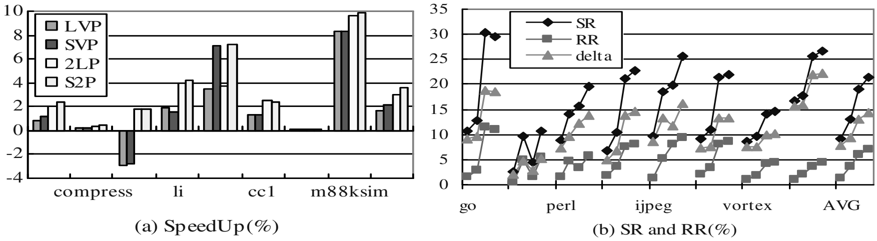


Fig. 5. Prediction Performance of Four Predictors Using Conf8, T_{verify}=0, $T_{reissue}$=0

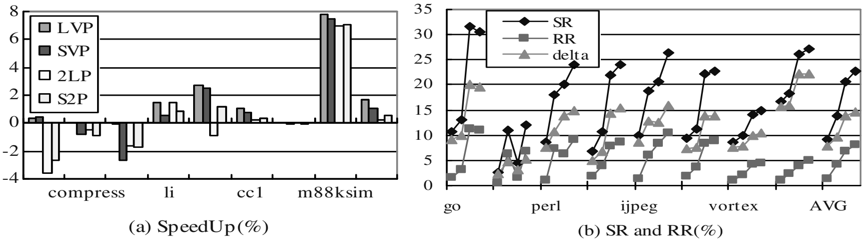


Fig. 6. Prediction Performance of Four Predictors Using Conf8, T_{verify}=0, $T_{reissue}$=1

4.3.3 Prediction Performance Using Conf8_bP

Figure 7 and Figure 8 demonstrate the simulation results of using Conf8_bP for $T_{reissue}$ = 0 and $T_{reissue}$ = 1 respectively. It is disappointing to find that all value predictors can hardly obtain performance benefits in both cases. One possible explanation may be obtained from figure 7(b) and 8(b). For many programs, the reissue rate is greatly higher than delta, which means most speculatively executed instructions get incorrectly performed. To manage reissue rate under an acceptable level is of great importance in value prediction design.

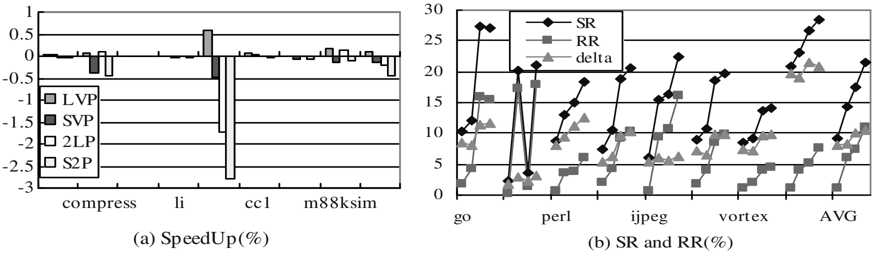
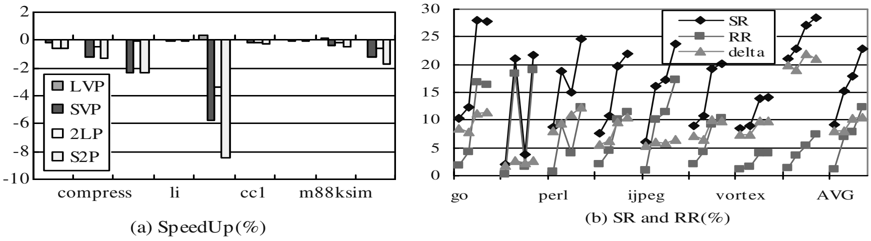


Fig. 7. Prediction Performance of Four Predictors Using Conf8_bP, T_{verify}=0, $T_{reissue}$=0



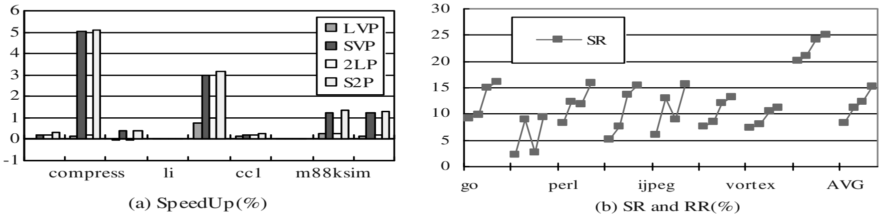
(a) SpeedUp(%)

(b) SR and RR(%)

Fig. 8. Prediction Performance of Four Predictors Using Conf8_bP, T_{verify}=0, $T_{reissue}$=1

4.3.4 Prediction Performance Using Conf8_bPvP

In this subsection, we will check the performance impact of perfect value predictors. All predictors are configured to make predictions only if the predicted value equals to the final result. Thus all speculatively executed instructions need not be reissued. Results are presented in Figure 9.



(a) SpeedUp(%)

(b) SR and RR(%)

Fig. 9. Prediction Performance of Four Predictors Using Conf8_bPvP, T_{verify}=0, $T_{reissue}$=1/0

We can see that the performances of programs are improved mostly, especially for *compress*, *ijpeg* and *m88ksim*. However, for *li* and *vortex*, negligible benefits are obtained. And for *perl*, negative effect is taken. One possible explanation is that under such ideal simulation environments where branch prediction is perfect and instruction window is very large, the original hardware itself is enough to obtain the best performance through instruction rearranging. Also the speculatively executed instructions are useful only if they are on the critical path. Thus the high speculation rate does not mean high performance gains.

5 Summary and Future Works

In this paper, we have discussed the pipelined execution of value speculation with different misprediction penalties and different architecture configurations. And the performance impacts of different data value predictors are studied in detail. We find that with different misprediction penalties and implementation issues, these predictors showed different characteristics. Some value predictors may even cause performance degradation under some configurations. And to our surprise, complicated hybrid value predictor, due to its high reissue rates, obtains lower IPC than simpler schemes in some of the experiments. These observations are of great importance in future value prediction research.

In the experiments, the speedup obtained with value predictor is moderate; benefits are even negligible under ideal environments with large window size, high issue width and perfect branch predictions. Reasons include: 1) unlike branch prediction, in our six-stage pipeline implementation, the benefit of correct value prediction is moderate; 2) the high reissue rate impairs the effect of value prediction; 3) many correct speculative executions may not be useful, because they are not on the critical path; 4) the relative low fetch bandwidth may also impairs the performance [9].

There are several directions for future work. One important work is to study the impact of value prediction in future designs. As deeper pipeline will be widely used, the benefit of correct value prediction may be larger. How to combine value prediction with architectures that exploit higher level parallelism [10, 11, 13], such as trace processor, to achieve higher speedups is still not clearly studied. Another direction would be to design better confidence and filtering mechanisms [8, 14] to limit unnecessary value predictions. Moreover, exploiting new type of value predictions, such as global stride locality [12], may improve the performance ulteriorly.

Acknowledgements. Thanks to the reviewers for their insight and helpful suggestions. This research is supported by the National Natural Science Foundation of China, NO. 90307001.

References

1. M. H. Lipasti, C. B. Wilkerson, and J. P. Shen. Value Locality and Load Value Prediction. Proceedings of VIIth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS- VII), 1996
2. M. H. Lipasti and J. P. Shen. Exceeding the Dataflow Limit via Value Prediction. Proceedings of 29th International Symposium on Microarchitecture (MICRO-29), 1996: 226-237
3. M. H. Lipasti and J. Shen. Exploiting Value Locality to Exceed the Dataflow Limit. International Journal of Parallel Programming, Vol. 28, No. 4, August 1998: 505-538
4. K. Wang and M. Franklin. Highly Accurate Data Value Prediction using Hybrid Predictors. Proc. of the 30th Annual International Symp. on Microarchitecture, Dec. 1997: 281-290
5. B. Rychlik, J. Faitl, B. Krug, J. P. Shen. Efficacy and Performance Impact of Value Prediction. Proceedings of International Conference on Parallel Architectures and Compilation Techniques, 1998
6. J. Gonzalez and A. Gonzalez. The Potential of Data Value Speculation to Boost ILP. 12th International Conference on Supercomputing, 1998
7. Y. Sazeides. Modeling value prediction. 8th International Symposium on High Performance Computer Architecture (HPCA-8), 2002
8. B. Calder, G. Reinman, and D. Tullsen. Selective Value Prediction. Proceedings of the 26th Annual International Symposium on Computer Architecture, June 1999

9. F. Gabbay and A. Mendelson. The Effect of Instruction Fetch Bandwidth on Value Prediction. 25th International Symposium on Computer Architecture (ISCA), 1998: 272-281
10. Y. F. Wu, D. Y. Chen, and J. Fang. Better Exploration of Region-Level Value Locality with Integrated Computation Reuse and Value Prediction. ISCA-28, July 2001
11. S. J. Lee, Y. Wang, and P. C. Yew. Decoupled value prediction on trace processors. In 6th International Symposium on High Performance Computer Architecture, Jan. 2000: 231-240
12. H. Zhou, J. Flanagan, and T. M. Conte. Detecting Global Stride Locality in Value Streams. The 30th ACM/IEEE International Symposium of Computer Architecture (ISCA-30), June 2003
13. S. J. Lee, P. C. Yew. On Some Implementation Issues for Value Prediction on Wide-Issue ILP Processors. IEEE PACT 2000: 145-156
14. R. Bhargava, L. K. John. Performance and Energy Impact of Instruction-Level Value Predictor Filtering. First Value-Prediction Workshop (VPW1) [held with ISCA'03], June 2003: 71-78
15. D. C. Burger, T. M. Austin. The SimpleScalar Tool Set, Version 2.0. Technical Report CSTR-97-1342, University of Wisconsin, Madison, June 1997.
16. S. J. Lee. Data Value Predictors. http://www.simplescalar.com/

Heterogeneous Networks of Workstations

SunHo Baek[1], KyuHo Lee[1], JunSeong Kim[1], and John Morris[1,2]

[1] School of Electrical and Electronics Engineering,
Chung-Ang University, Seoul 156-756, Korea
[2] Department of Electrical and Electronic Engineering,
The University of Auckland, New Zealand

Abstract. Parallel systems built from commodity CPUs and networking devices (Networks of Workstations or NoW) are easily and economically constructed. Rapid technological change means that as elements are added to or replaced in such a system, it will inevitably become heterogeneous.

This paper examines issues in effectively using all the processing power of a heterogeneous NoW system. In particular, it examines the use of two different parallel programming models: MPI, a popular message passing system and Cilk, an extension to C with dataflow semantics. We consider both the performance of the two systems for a set of simple benchmarks representative of real problems and the ease of programming these systems. We show that, in many cases, Cilk both performs better than MPI and, additionally, that it requires less programming effort to write programs that exploit the full power of a heterogeneous system.

1 Introduction

Demand for personal computers has led to production volumes that have meant that the most cost-effective way to obtain raw processing power is to buy commodity personal computers. The de facto requirement that most personal computers are connected to networks of some type has also meant that networking hardware (interfaces, routers, switches, *etc.*) is also readily available and economic. Thus potentially powerful, but economic, parallel processors or Networks of Workstations (NoWs) can be built from commodity PCs. In this paper, we focus on systems built from commodity components containing only very small numbers (say 1-2) processors on each system bus. Thus we assume either no or very limited shared memory.

Rapid and continual technological advances also mean that more powerful processors are almost continually available as manufacturers release new, faster processor chips. Volume production of personal computers allows manufacturers to rapidly incorporate the faster chips into systems - in fact, for many manufacturers the ability to release a system incorporating a faster processor as soon as the new CPU chip is available is vital for survival. Thus, unless a system has been put together from identical processors over a very short period of time, it will inevitably be a heterogeneous one: as processors are added or replaced,

P.-C. Yew and J. Xue (Eds.): ACSAC 2004, LNCS 3189, pp. 426–439, 2004.

the downward pressure on prices means that more cost-effective processors are available. After quite a short period of time, it may even become impossible to locate processors matching the original components of a NoW: in today's market, processors that lag behind the state-of-art quickly become 'obsolete' - despite having considerable processing capability.
If the parallel processing run-time system (which could be a message passing one such as MPI[1], one which creates an illusion of shared memory[2] or the RTS supporting Cilk's dataflow mode of operation[3]) naively distributes equal amounts of work to each processor in a heterogeneous NoW, then the presence of a final barrier at which results from all sub-calculations are gathered together to generate the ultimate result leads to an n-processor system equivalent, at best, to n of the slowest processors. Parallel processing overheads will inevitably reduce the NoW system's power to considerably less than this. This obviously represents a considerable waste of resources in a system where the power of individual processors may vary by a factor of 10 or more. Thus, the ability to effectively exploit the more powerful processors in a heterogeneous system is important: it allows efficient systems to be built, maintained and augmented. Heterogeneity may come from sources other than raw CPU clock speed. Cache sizes and speeds, the mix of processing units in a superscalar, bus speeds, peripheral interface efficiency (particularly network cards) and memory speed and capacity may all change the 'power' of one element of a NoW relative to another. Relative powers are also problem dependent, so that simple load balancing schemes that rely on knowledge of relative processing powers are unlikely to be effective over a wide range of problems - as well as requiring significant set up or calibration effort.

1.1 Message Passing Run-Time Systems

A message passing run-time system provides one basic capability - the ability to send a message from one process to another. Thus the RTS needs to provide `send` and `receive` primitives. Generally, these will provide synchronization also: a process will block on a `receive` until the sending process has executed the `send`, synchronizing two threads of computation as well as transferring data. Providing very low level semantic support, message passing systems require more programmer effort to manage data, threads of computation and synchronization. This is usually reflected in higher program complexity - confirmed by our measurements in Section 3.6.

1.2 Cilk's Dataflow Model

Although Cilk is an extension of a weakly typed programming language, it has a strong semantic model. Only when all their data is available, do threads 'fire' (become ready for execution) Thus Cilk programs are data-driven as far as the main computational units - the threads - are concerned and control-driven within each thread. A quite small extension to C, Cilk is easy to learn but its safety (based on the strict dataflow model) is compromised by the ability to use C pointers (particularly global ones). On the other hand, it is able to leverage extensive experience in optimizing C compilers. A safer version of Cilk - Dataflow Java - has been implemented and has been demonstrated to run in fully heterogeneous environments consisting of processors running different operating systems[4], but its performance could not match that of C-based Cilk.

2 Experiments

Programs for several simple problems were written in both C using an MPI library[5] and Cilk. Some of the problems had parameters other than the basic problem size which could be used to change the characteristics (numbers and sizes of messages needed to transfer data between processors) and thus fully explore the relative efficiencies of various strategies.

The essential characteristics of the problems used are described in the following sections. The major basis for comparison was the speedup attainable for a suitably large (and therefore interesting!) problem.

In addition to measuring performance, we assessed program complexities by counting lines of code (separated into 'core' algorithm code and support code, *e.g.* code to manage work queues) and variables used. We also examined the effort needed to change a basic problem solution to better use the heterogeneous environment and report the gains resulting from these changes.

MPI lends itself to simple work distribution schemes in which fixed blocks of compuation are sent to individual processing engines (PEs). Prediction of the optimum block size in a heterogeneous NoW environment is not a simple task: we measured the relative performance of the PEs in our testbed when solving the test problems and observed a range of relative 'powers' (*cf.* Table 1). Thus any assumption of fixed powers is treacherous. We partly overcame this problem by probing: each PE was asked to solve the problem individually and the performance factors used to determine load balancing factors. Using these factors produced, as expected, much better speedups: without them (*i.e.* with equal work loads) the system was essentially constrained to be n copies of the slowest processor. However, conflicts for the communications network, queuing of large packets, *etc.*, mean that simple pre-determined loading factors are not ideal: better results can be obtained with dynamic load balancing schemes. The Cilk RTS's 'work-stealing' mode of operation is well suited to this and can lead to near-optimum use of the system (*cf.* Figure 3). It's ability to create work on slave processors further helps load balancing. To achieve a similar effect with MPI, we built a small work packet management suite of routines, which were sufficiently generic that they could be used with many problems. They provided a simple queue to which work descriptors could be added by the work generator (usually the master PE) and extracted for despatch to slaves as needed. Because this suite could have been transferred to a library and used without further alteration for many problems, its code was tallied separately in the code volume comparisons (see Table 2).

2.1 Fibonacci

We used the naive recursive algorithm taught in every course on recursive programming and then dumped by courses on algorithms and complexity! It is simple, provides a simple demonstration of the difficulties of MPI *vs* Cilk programming and can be parameterized to generate a wide range of loads on the underlying communications system by adjusting the depth at which new parallel computation threads are spawned.

2.2 Travelling Salesman Problem (TSP)

A classic ‘hard’ problem, TSP requires a search through $n!$ permutations of a list. As with Fibonacci, a simple recursive program solves the problem. In parallel, good speed-ups are trivially obtained with little communication overhead by despatching sub-tours to be evaluated to individual PEs. The cost matrix is moderately large ($\mathcal{O}(n^2)$), but it only needs to be distributed to each PE once. Sub-tour descriptors are short integer lists requiring short messages. An optimization distributes the cost of the best tour obtained so far to all PEs, enabling unprofitable searches (a sub-tour is already longer than the best tour found so far) to be abandoned. This optimization dramatically speeds up execution although it generates more messages to stress the RTS/communication system.

2.3 N-Queens Problem

Another ‘hard’ problem, $N - Queens$ requires an exhaustive search to find all the possible placements of non-threatening queens on an $N \times N$ chess-board. It has no set-up costs (*e.g.* distribution of a cost matrix for TSP) and spawns $\mathcal{O}(N)$ threads recursively at every stage: thus it has a ‘pure’ recursive code solution. It seeks to find all the possible placements, so does not have a cut-off optimization like TSP. Parallel overhead can be reduced in the Cilk version by introducing a sequential depth - or point at which no more threads are spawned (as with Fibonacci).

2.4 Matrix Multiplication

A simple, but important problem, matrix multiplication (MM) is trivially parallelized by dividing one of the operand matrices into bands and computing sub-products on each PE. The other operand matrix is distributed to all PEs in the initialization phase. It has $\mathcal{O}(n^2)$ communication cost and $\mathcal{O}(n^3)$ computation cost: with modern fast CPUs and relatively slow networks, this implies that speedup will only be observed with matrices large enough to overcome fixed overheads. Large blocks of data are also transmitted: in our test problem, a 6Mbyte block of data must be transferred to each PE at startup. Note that the algorithm tested here starts with the two operand matrices on a ‘master’ PE and finishes with the product on the same PE. Other work assumes distributed operand and product matrices: for example, Beaumont *et al.*’s study showed how to find an effective allocation in a heterogenous environment[6].

2.5 Finite Differencing

This problem solves Laplace's equation in a discrete form by updating a cell with the average of the four nearest-neighbour cells. One application is determination of heat flow through a conducting plate. The plate is represented by a rectangular matrix of cells. Each PE is assigned a band of cells (a set of rows of the matrix) to solve. The problem is solved iteratively: it converges when the cell values in successive iterations change by less than some accuracy criterion. Rows of boundary cells are shared between nearest neighbours: in each iteration each PE only needs to update its two neighbours with new cell values. This allows several synchronization schemes. The simplest one uses a 'master' PE. Each PE sends a signal to the master when its neighbours' boundaries have been updated[1]. It also sends a signal to the master which advises whether the band has converged or not. When the master has received all the synchronization signals, if any of the bands is yet to converge, the master signals each PE to calculate one more iteration. An alternative scheme synchronizes between nearest neighbours only, allowing a degree of asynchronous processing - processors may overlap computation and boundary value communication more efficiently. A further scheme allows the individual processors to run completely asynchronously: each PE will commence a new iteration whether boundary updates have arrived from neighbours or not. Boundary updates are despatched to neighbours when they are available.

Table 1. Characteristics of our NoW testbed

Name	Clock	Memory	Relative	OS
	MHz	Mbytes	Power	Linux
ant1-6	400(P-II)	128	1.00	v7.3
ant7-8	450(P-III)	128	1.07-1.10	v7.3
ant9-10	800(P-III)	128	1.94-2.45	v9.0
ant11-16	800(P-III)	128	1.74-2.44	v7.3

2.6 Test Bed

Following usual practice when building NoW systems, we begged and borrowed a collection of PCs of varying ages and capabilities. Components were replaced as necessary to build working systems, not to produce a homogeneous system. The memory of each system was increased to at least 128Mbytes. Again, no attempt was made to ensure equality or fairness - except with respect to the minimum amount of memory. None of the programs had working sets large enough to cause significant amounts of paging. None of the benchmark programs used the disc system for anything except initial program loading (which was not included in

[1] An error in Cilk's specification was discovered when this program was written: see Appendix A.

the timed results) and trivial amounts of result logging. Thus the performance (or lack of it) of individual discs did not affect the results. Note that the Cilk program distribution system adds PEs to the system in the order in which they responded to the initial broadcast. No attempt was made to control this order as it mimics a real heterogeneous system in which PEs have been upgraded as old components were replaced. One of the slower processors was always used as the master and speedups are always referred to it. This arrangement is the optimal one for MPI where the simplest programming solution often uses a master PE for synchronization and data distribution but not for any significant amounts of computation.

3 Results

With a heterogeneous NoW, there are two possible bases for measuring speedup:

- simply counting PEs (*i.e.* making them all equal) so that for 16 PEs, a speedup of 16 is ideal, and
- weighting them by their 'raw' powers, *cf.* Table 1.

Thus two target speedup lines have been placed on speedup graphs - 'Basic speedup' assumes that all PEs are equal and 'Possible speedup' considers individual PE powers. In our system, the potential speedup for 16 PEs relative to the slowest PE was ~ 25: it is slightly problem dependent due mainly to differing cache utilization.

3.1 Fibonacci

This problem has the finest grain parallelism of all those tested: thus parallel overheads are quite high. To contain these overheads, the level at which new threads are spawned was controlled in both cases. For Cilk, this involved four additional lines of code. In the MPI version, the generation of sub-tasks for distribution has to stop at some level - otherwise most of the work goes into generating them! Thus no speedup can be observed at all without controlling this level, and so it is built into the general programming overhead. Figure 1 shows the results for the optimum spawning level for each system size (*i.e.* number of PEs). Cilk generally shows a slight performance edge: the speedup difference is very small for small numbers of PEs (with MPI occasionally being slightly faster), but for larger systems, Cilk consistently produces measurably better results.

Noting that the spawning level essentially controls the computational cost of a thread: a lower spawning level generates fewer, more expensive threads, we examined the effect of the spawning level on performance. Figure 2 shows that although the returns are diminishing, a large number of smaller cost threads increases performance. For $fib(43)$, $sp_{level} = 13 \rightarrow 8192$ threads - or considerably more than would be expected to be needed to keep 16 PEs busy. A simple model which ensured that the slowest processor received, say 10, packets and the faster

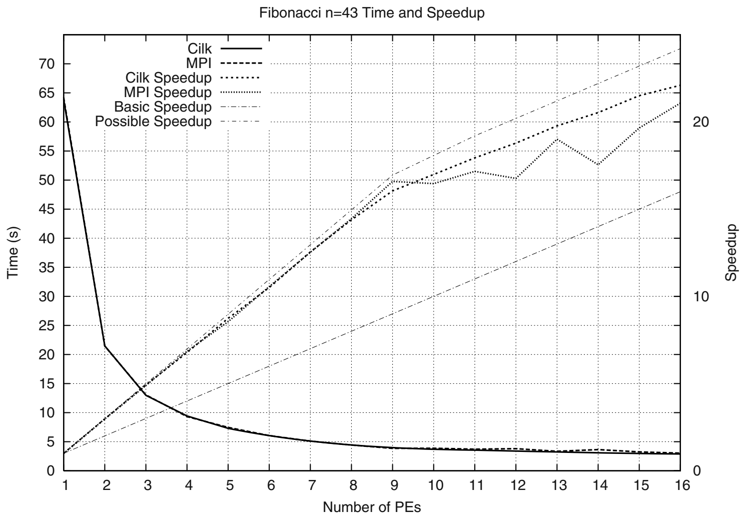


Fig. 1. Times and speedup for Fibonacci

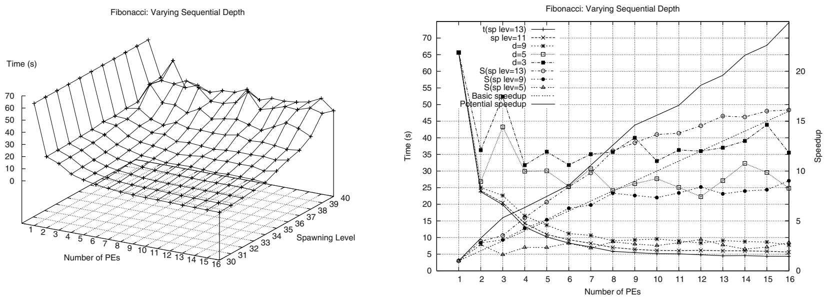
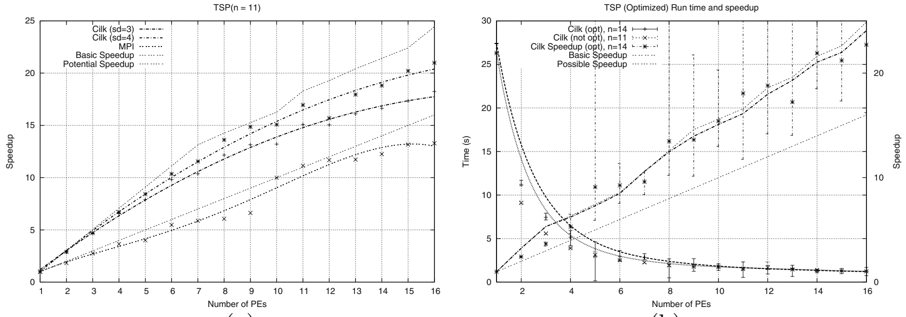


Fig. 2. Fibonacci: Effect of varying the spawning level

ones proportionately more - so that a time within 10% of the optimum was obtained requires only 250 packets for our system. However, going from $sp_{level} = 11 \rightarrow sp_{level} = 13$ (or 2048 → 8192 threads) produces a 20% improvement in time for 16 PEs (*cf.* Figure 2(b)).
With large numbers of threads (representing small work quanta) spawned, idle-initiated work-stealing in the Cilk RTS ensures that the heterogeneous system is efficiently used. Small work quanta are always available to a PE which becomes idle. However, if every PE is busy, they are executed locally with no communication overhead.

3.2 Travelling Salesman Problem

The unoptimized version of TSP shows excellent speedups because each spawned thread at the top level represents a large amount of work for a small communication effort. The results are shown in Figure 3(a). For Cilk, speedups approached the potential speedup. Note again the importance of creating sufficient threads, as Figure 3(a) shows, significant improvement is obtained by increasing the sequential depth from 3 to 4, which increases the number of threads generated from 990 to 7920. The unoptimized Cilk program is particularly simple: the core has only two parallel statements, one to spawn the initial thread and the other to spawn threads to work on sub-tours. Two `sync` statements are also needed[2]. The MPI version did not distribute the generated sub-computations in as efficient a way as the Cilk program and its speedup suffered.



(a) (b)

Fig. 3. TSP (a) unoptimized, $n = 11$ (b) $n = 14$(optimized) compared with $n = 11$(unoptimized)
Speedup - maximum speedup obtainable for n equal machines
Possible speedup - maximum speedup considering individual PE capabilities
Note the large variation in times for the optimized algorithm - reflecting the number of asynchronous processes present! Results from this optimization are very sensitive to the the order in which sub-tours are evaluated leading to the occasional ‘super-linear’ speedups seen here.

An optimization to the simple TSP program broadcasts the best tour found so far to all other processors. PEs use this value to cut off unprofitable searches in which a sub-tour cost is already greater than the best tour seen already. Whilst not altering the fundamental hard nature of the problem, it dramatically increases the rate at which the search is carried out by cutting off large portions of the search space: we observed speedups of nearly three orders of magnitude using it - enabling a 14-city problem to be solved in about the same time as the unoptimized version solves a 11-city one. However, it does generate a large number of additional messages which affects the speedups shown in Figure 3(b).

[2] The Cilk compiler could infer the need for these from a simple dataflow analysis of the code (detecting definition-use pairs), but its authors presumably considered that a programmer could do a better job of deciding the optimum placement of the minimum number of `sync`’s.

Adding this optimization to the Cilk programs was simple: a global variable was created which stores the cost of the best tour found on any processor. When a PE finds a better tour, it spawns a thread to update this value on each other PE. Several PEs might be doing this at any time, but Cilk's run-time model ensures that once a thread is started, it runs to completion, so that all these update threads are run in some order on each PE. The thread simply checks to ensure that it does not replace a better minimum cost that some other PE may have found. Two spawn statements (one ensures that the global best cost is initialized correctly on each PE, the other updates the best cost) and 6 lines of additional code suffice to produce a dramatic speedup.

3.3 N-Queens Problem

Cilk's ability to generate large numbers of threads ensures good utilization of all the PEs and speedups which are close to optimal and slightly better than MPI's. As with the other MPI implementations, a fixed number of computations were allocated to each PE. Using relative PE powers to determine how many computations are assigned to each PE is complicated by the fact that spawned computations represent different amounts of work - some positions are declared impossible at a high level in the tree and generate no further computation. In Cilk, this does not affect efficient load balancing because if a large amount of work is stolen by a slow PE, it will spawn several threads, some of which may be stolen by faster PEs. Again, larger numbers of work packets benefited the MPI implementation and brought its performance close to that of the Cilk version.

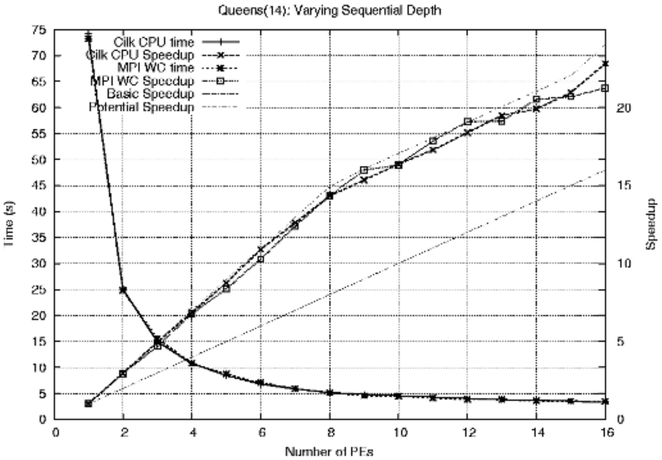


Fig. 4. Times and speedup for N-Queens $n = 14$

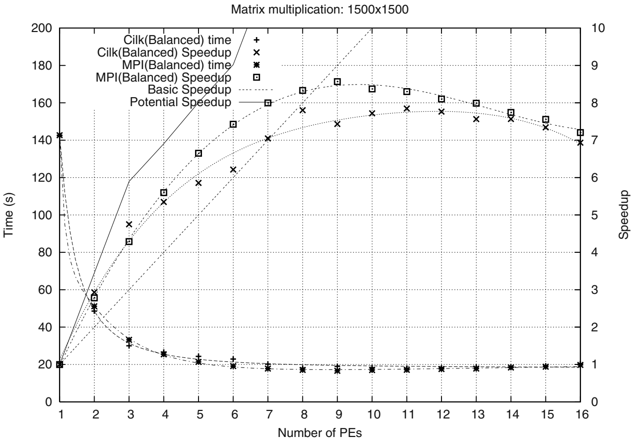


Fig. 5. Times and speedup for Matrix Multiplication

3.4 Matrix Multiplication

Since matrix multiplication is very regular with simple, predictable load patterns, it might be expected that MPI's direct message passing would have lower overheads and produce better results. However, in a previous study[7], Cilk performed better: this was attributed to Cilk's synchronize-in-any-order dataflow model resulting in cheaper synchronization for the final result. We repeated these experiments for the current heterogeneous system with several different approaches to load balancing: equal loads and balanced (or power-weighted) loads (with no work stealing). As expected, for this problem, the equal loads approach performs poorly when individual PE powers vary by a factor of ~ 2.44 and so we present only balanced load results.
Cilk programs exhibited slightly higher speedups with small numbers of PEs ($n \leq 3$). With a small number of large messages, out of order completion has relatively more effect and Cilk performs slightly better. However as the individual slice computation time and message size drops, delays introduced by the strict read order imposed by our MPI program become less significant and MPI's simple direct message-passing model produces higher speedups. Speedups reached peaked at 11 PEs with Cilk (*vs* 9 PEs with MPI): reflecting the increased competition for the common network in the initialization and completion stages of the algorithm. Thus for small systems, Cilk was able to perform better, but MPI's best performance was significantly better than Cilk's.

3.5 Finite Differencing

The simple, regular communication patterns required for this problem match MPI's capabilities well and this is reflected in the superior performance of MPI using the simplest master synchronization scheme. Even an elaborate distributed

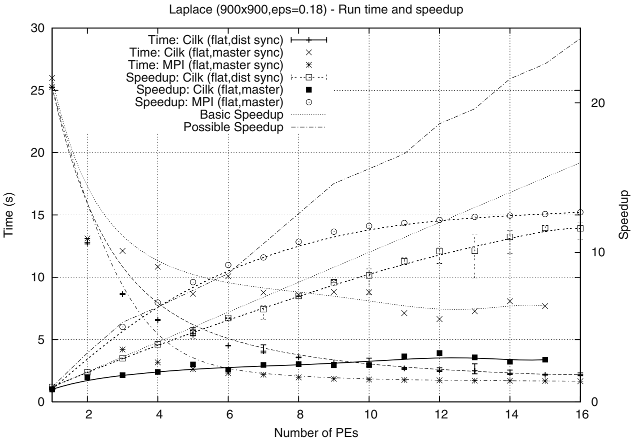


Fig. 6. Time and speedup for several variants of finite differencing program

synchronization scheme in which each PE only synchronized with its two neighbours did not allow Cilk to match the MPI results. Comparing the Cilk results for the two synchronization styles is a textbook example of the cost of synchronization in parallel systems! We did not implement the same scheme in MPI[3]. Although there is a simple dataflow solution to this problem, it is usually avoided as it requires unnecessary data copying. Our Cilk implementation used Cilk's ability to provide Active Messages[8] for the boundary updates and global pointers to the data arrays on each PE. This is a significant departure from Cilk's underlying dataflow model, so it is not surprising that its performance suffers.

3.6 Program Complexity

Whilst they are far from perfect metrics, we used counts of lines of code (LOC) and variables to provide some estimate of relative programming difficulty. The counts in Table 2 are for the 'core' code that solves the problem. Code that generated the test data set and code used for instrumentation (timing or measuring other aspects of performance, *e.g.* counting iterations in Laplace) was not included. The tallies were divided into two groups: basic code needed to solve the problem (code that would be present in a sequential version of the same problem) and parallel overhead - code to initialize and configure the system, create threads, transmit data, synchronize, *etc. Disclaimer: software metrics is not an exact science!*[4] The figures provided here are very basic and simply provide an

[3] Our time was consumed by the absence of a 'kill' instruction (to invalidate all spawned waiting threads when convergence has been reached) to support the Cilk RTS' requirement for all threads to fire and complete!

[4] Our counts were generated by hand: we considered that the additional effort required to remove instrumentation and diagnostic code in order to use a tool to obtain more sophisticated counts was unwarranted.

Table 2. Code complexities

Problem	Cilk				MPI					
	Algorithm		Overhead		Algorithm		Overhead		Lib	
	LOC	var	LOC	var	LOC	var	LOC	var	LOC	
Fibonacci	6	2	1	0						*a*
+seq depth	10	2	3	1	7	3	12	5	52	
TSP	20	13	9	0						*a*
+seq depth	23	13	9	1	18	10	14	6	59	
+opt	30	14	9	1						*b*
MM	9	6	27	21	9	6	11	6	51	
N-Queens	19	13	2	1						*a*
+seq depth	26	12	2	2	19	10	11	4	55	
Laplace										
master synch	17	10	42	44	15	6	18	2		
dist synch	30	5	20	10						*b*

Notes: (a) Sequential depth optimization is fundamental to MPI version.
(b) Not implemented.

indication of programming effort. A small amount of ‘noise’ - in the form of individual programming style - was also added by the several authors of the programs used. However, the lower parallel programming overhead of Cilk is clear: by providing a simple semantic model, it provides considerably more support for a programmer than MPI. It is also notable that some of the optimizations are trivially added to Cilk programs.

With MPI, in order to obtain good speedups, it was necessary to build simple lists of ‘work packets’ which were distributed to each PE. The code to manage these queues has been included in the column marked ‘Library’ in Table 2 because generic code for this could be placed in a library so that each program could only see `create, add` and `get` methods: effectively these methods would become part of the MPI library.

4 Conclusion

With the exception of the finite differencing program, Cilk versions of the programs were considerably easier to write. Implicit synchronization using a dataflow model means fewer statements directing parallel execution are needed. Most Cilk programs (fib, TSP and N-queens in the set used here) will run on any number of processors (including a single PE) without any explicit consideration of the current size of the NoW system. The program that we used for distributing Cilk programs to individual PEs simply counts the number of PEs responding to a broadcast request and the RTS uses this number when making work stealing requests: thus Cilk programs work well in ‘flexible’ environments - where the number of PEs may change from day to day. The implicit generation of large numbers of threads by Cilk programs also leads to superior speedup numbers in heterogeneous environments. In cases where explicit distribution of work to

PEs is appropriate because obvious simple distribution policies are readily coded (MM, Laplace), both Cilk and MPI require similar numbers of statements, *i.e.* MPI has no significant advantage when it might be expected to have one.
When a strict dataflow model requires excessive data copying (Laplace), MPI both allows (slightly) simpler implementations and provides better performance. We note that both systems are C-based, so that hybrid programs which, for example, used Cilk's dataflow model when it was effective and MPI's 'direct' transfer of data from source to destination when it provided better performance, could readily be put together.
We speculate that the best way to improve the performance of MPI programs - particularly in heterogeneous environments - is to simulate Cilk's idle-initiated work stealing mode of operation: generating large numbers of relatively fine-grained work packets which are distributed to PEs on demand, *i.e.* as they become idle. However to fully emulate Cilk's flexibility - threads can be spawned and stolen from any PE - would require duplicating a large part of the Cilk RTS. It would seem more efficient to simply use the Cilk RTS!

Acknowledgements. John Morris was supported by the Foreign Professors Invitation Program of the Korean IT Industry Promotion Agency at Chung Ang University in 2003-4.

References

1. Snir, M.: MPI: The complete reference. MIT Press, MA: Cambridge, USA (1996)
2. Amza, C., Cox, A.L., Dwarkadas, S., Keleher, P., Lu, H., Rajamony, R., Yu, W., Zwaenepoel, W.: TreadMarks: Shared Memory Computing on Networks of Workstations. IEEE Computer **29** (1996) 18–28
3. Blumofe, R.D., Joerg, C.F., Kuszmaul, B.C., Leiserson, C.E., Randall, K.H., Zhou, Y.: Cilk: an efficient multithreaded runtime system. In: PPoPP'95, Santa Barbara (1995)
4. Lee, G., Morris, J.: Dataflow Java: Implicitly parallel Java. In: Proceedings of the Australasian Computer Systems Architecture Conference. (1998) 42–50
5. Gropp, W., Lusk, E., Doss, N., Skjellum, A.: A high-performance, portable implementation of the MPI message passing interface standard. Parallel Computing **22** (1996) 789–828
6. Beaumont, O., Boudet, V., Rastello, F., Robert, Y.: Matrix multiplication on heterogeneous platforms. IEEE Transactions on Parallel and Distributed Systems **12** (2001) 1033–1051
7. Tham, C.K.: Achilles: A high bandwidth, low latency, low overhead network interconnect for high performance parallel processing using a network of workstations. PhD thesis, The University of Western Australia (2003)
8. von Eicken, T., Culler, D.E., Goldstein, S.C., Schauser, K.E.: Active Messages: a Mechanism for Integrated Communication and Computation. In: Proceedings of the 19th Annual International Symposium on Computer Architecture, Gold Coast, Australia. (1992) 256–266

A Cilk Signals

When writing the master synchronous version of the Laplace program, an error was discovered in the implementation of Cilk. The language specifies a `signal` which can be used to fire threads without transferring a data value. The syntax adopted allows the number of signals needed to fire a thread to be specified when it is spawned, *e.g.*

```
  cont signal s;
spawn SynchThread( ?s{n}, ... );
```

The thread `SynchThread` will be fired after n signals have been sent to it. The continuation `s` may be sent as an argument to several threads:

```
cont signal s;
for(pe=0;pe<m;pe++)
spawn CompThread( s, ... )@pe;
```

Note that the cardinality of the signal is not passed to `CompThread`: each instance of it may send as many signals as it likes to `SynchThread`. As long as `SynchThread` receives n signals, it will eventually fire. This presents no problem unless `CompThread` passes the continuation to a thread which is executed (by stealing or explicit allocation) on another processor. When threads with continuations migrate, Cilk versions from 2.0 on, have created a **stub** on the executing processor and altered the continuation to address this stub. However, the stub does not know how many signals it will be expected to pass to the original thread, so it destroys itself after sending one signal. In the Laplace program, each iteration sends *two* signals back to the master's synchronization thread when the boundaries of the two neighbouring PEs are updated.
We solved this problem by

- defining a new `ArgCont` type which was used to transfer a continuation to another thread and
- creating a simple `SendSignal` thread which was used to transfer the signal back to the master processor.

Use of the `ArgCont` type prevented the Cilk pre-processor from recognizing the parameter as a continuation and creating a stub. The `SendSignal` thread was executed on the master - where the original continuation had been created and was valid.

Finding High Performance Solution in Reconfigurable Mesh-Connected VLSI Arrays

Jigang Wu and Thambipillai Srikanthan

Centre for High Performance Embedded Systems,
School of Computer Engineering,
Nanyang Technological University, Singapore, 639798
{asjgwu, astsrikan}@ntu.edu.sg

Abstract. Given an $m \times n$ mesh-connected VLSI array with some faulty elements, the reconfiguration problem is to find a maximum-sized fault-free sub-array under the row and column rerouting scheme. This problem has already been shown to be NP-complete. The power awareness problem of the reconfigurable array is first proposed in this paper. A heuristic algorithm has been presented for the same. The algorithm is simple but efficient. The performance of the proposed algorithm is more powerful than that of the older algorithm, without loss of harvest.

Keywords: Degradable VLSI array, reconfiguration, heuristic algorithm, fault-tolerance, NP-completeness.

1 Introduction

Area-time efficiency is one of the major considerations in VLSI designs. In recent years, the growth of personal computing devices (portable computers and real time audio and video-based multimedia products) and wireless communication systems (personal digital assistants and mobile phones) has forced designers to make high performance systems that consume less power. This necessitates the need to minimize the number of switches employed to implement the interconnection between two processing nodes during rerouting. Hence, degradable arrays that involve minimal number of switches will provide for higher performance while improving the overall reliability.

Mesh is one of the most thoroughly investigated network topologies for multiprocessors systems. It is of importance due to its simple structure and its good performance in practice and is becoming popular for reliable and high-speed communication switching. It has a regular and modular structure and allows fast implementation of many signal and image processing algorithms. With the advancement in VLSI technologies, integrated systems for mesh-connected processors can now be built on a single chip or wafer. As the density of VLSI arrays increases, probability of the occurrence of defects in the arrays during fabrication also increases. These defects obviously affect the reliability of the whole system. Thus, fault-tolerant technologies must be employed to enhance the yield and reliability of this mesh systems.

P.-C. Yew and J. Xue (Eds.): ACSAC 2004, LNCS 3189, pp. 440–448, 2004.

Reconfiguration in a mesh structure has been investigated extensively, *e.g.*, [1-8] for redundancy approach and [9-12] for degradation approach. However, no previous work has been carried out in order to provide high performance sub-mesh for degradable VLSI arrays with faults. This paper proposes the problem of finding a fault-free sub-array which demonstrate higher performance in a two-dimensional mesh-connected VLSI arrays. A simple but efficient heuristic algorithm is proposed, which has higher performance requirements, while maintaining the same harvest.

2 Preliminaries

Let *host array* H be the original array obtained after manufacturing. Some of the elements in this array may be defective. A *target array* T is a fault-free sub-array of H after reconfiguration. The rows (columns) in the host array and target array are called the *physical row (columns)* and *logical rows (columns)*, respectively.

In this paper all the assumptions in architecture are the same as that in [9-12]. Neighboring elements are connected to each other by a four-port switch. A target array T is said to *contain* $\{R_1, R_2, \cdots, R_k\}$ if each logical column in T contains exactly one fault-free element from each of the rows. The previous problem and the related algorithms are as follows.

Problem $\mathcal{RP}$: *Given an* $m \times n$ *mesh-connected host array, find a maximal sized target array under the row and column rerouting scheme that contains the selected rows.*

The problem $\mathcal{RP}$ is optimally solved in linear time. Low proposed an algorithm, called *Greedy Column Rerouting (GCR)*, to solve $\mathcal{RP}$[10, 11]. Let $col(u)$ and $col(v)$ denote the physical column index of the element u and v, respectively. All operations in *GCR* are carried out on the adjacent sets of each fault-free element u in the row R_i, defined as

$Adj(u) = \{v : v \in R_{i+1},\ v \text{ is fault-free and } |col(u) - col(v)| \leq 1\}$.

The elements in $Adj(u)$ are ordered in increasing column numbers for each $u \in R_i$ and for each $i \in \{1, 2, \cdots, k-1\}$.

GCR constructs the target array in a left-to-right manner. It begins by selecting the leftmost fault-free element, say u, of the row R_1 for inclusion into a logical column. Next, the leftmost element in $Adj(u)$, say v, is connected to u. This process is repeated until a logical column is fully constructed. In each iteration, *GCR* produces the current leftmost logical column. A detailed description of *GCR* can be found in [11]. Theorem 1 describes the properties of the *GCR* algorithm.

Theorem 1. *GCR* solves the problem $\mathcal{RP}$ in linear time and produces the maximal sized target array[11].

As shown in Fig. 1, there are 6 possible types of link-ways for a target array. They can be categorized into two classes based on the number of the switches used. In this paper, a link-way that uses only one switch to connect neighboring elements in a target array is called a *regular link-way*, while a link-way using two switches is called an *irregular link-way*. In Fig. 1, (a) and (d) are regular link-ways, but the others are irregular. (a), (b) and (c) are used for row rerouting, while (d), (e) and (f) are used for column rerouting. Obviously, the smaller the number of irregular link-ways the target array has, the lesser is the system delay, and therefore, the higher the performance.

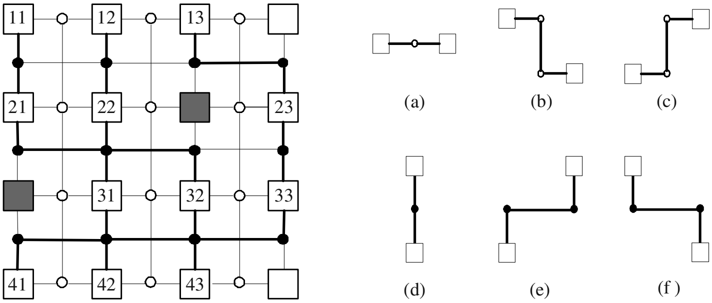


Fig. 1. A target array and its possible link-ways.

In this paper, the maximal sized target array with the minimal number of irregular link-ways is called the high performance target array or high performance solution. The problem that we proposed is described as follows.

Problem $\mathcal{HP}$: *Given an $m \times n$ mesh-connected host array, find a high performance solution under the row and column rerouting scheme that contains the selected rows.*

We aim to propose a heuristic algorithm to solve this optimization problem in this paper.

Given a host array H of size $m \times n$, let U be the set of logical columns that pass through each of the rows. Thus, each logical column in U contains exactly one fault-free element from each of the rows. Furthermore, the ith element in each logical column resides in row R_i, for each $i = 1, 2, \cdots, m$. We define a partial order on the logical columns in U as follows.

Definition 1. *For any two logical columns, C_p and C_q in U, and $p \neq q$,*

1. *we say that $C_p < C_q$ if the ith element in C_p lies to the left of the ith element in C_q, for $1 \leq i \leq m$.*
2. *We say that $C_p \leq C_q$ if the ith element in C_p lies to the left of, or is identical to, the ith element in C_q, for $1 \leq i \leq m$.*
3. *We say that C_p and C_q are independent, if $C_p < C_q$ or $C_p > C_q$.*

Assume $\mathcal{B}_l$, $\mathcal{B}_r \in U$ and $\mathcal{B}_l \leq \mathcal{B}_r$. We use $\mathcal{A}[\mathcal{B}_l, \mathcal{B}_r]$ to indicate the area consisted of the fault-free elements bounded by $\mathcal{B}_l$ and $\mathcal{B}_r$ (including $\mathcal{B}_l$ and $\mathcal{B}_r$).

$\mathcal{A}[\mathcal{B}_l, \mathcal{B}_r)$ indicates the same area as above including $\mathcal{B}_l$ but not including $\mathcal{B}_r$. $\mathcal{B}_l$ and $\mathcal{B}_r$ are called the left boundary of $\mathcal{A}[\mathcal{B}_l, \mathcal{B}_r)$ and the right boundary of $\mathcal{A}[\mathcal{B}_l, \mathcal{B}_r)$, respectively.

3 Algorithms

Assume the solution of *GCR* is the target array T with k logical columns: $C_1, C_2, \cdots, C_k$. We revise each C_i, for $i = 1, 2, \cdots, k$. The proposed algorithm, denoted as *RGCR* revises the solution of *GCR* to obtain an approximate high performance solution. It starts from the logical column C_k, revises it and then revises $C_{k-1}, \cdots, C_1$ one by one. In each iteration, *RGCR* works in $A[\mathcal{B}_l, \mathcal{B}_r)$. In the first iteration, $\mathcal{B}_l$ is set to be C_k, and $\mathcal{B}_r$ is set to be a virtual column that lies to the right of the nth physical column of the host array.

In *RGCR*, a priority to each element $v \in Adj(u)$ is assigned according to the following function for each $u \in R_i$, and for each $i \in \{1, 2, 3, \cdots, m\}$.

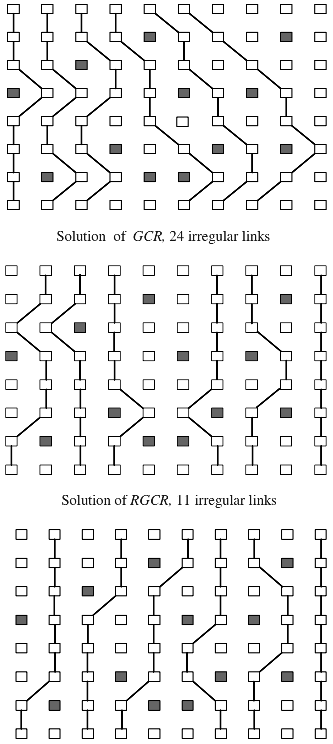
$$Pri(v) = \begin{cases} 1, \text{ if } & col(v) - col(u) = -1, \\ 3, \text{ if } & col(v) - col(u) = 0. \\ 2, \text{ if } & col(v) - col(u) = 1, \end{cases}$$

Suppose, the element $u \in C_k$ and it has the largest physical column index (it is the rightmost element) in the logical column C_k. In each iteration, the algorithm *RGCR* calls a procedure *REROUT* to revise the current logical column. *RGCR* starts off with $(0, col(u))$ as the start element for rerouting. The highest priority element in $Adj(u)$, say v, is connected to u. This process is repeated such that in each step, *REROUT* attempts to connect the current element v to the highest priority element of $Adj(v)$ that has not been previously examined. If *REROUT* fails in doing so, we cannot form a logical column containing the current element v. In that case, *REROUT* backtracks to the previous element, say w, that was connected to v and attempts to connect w to the highest priority element of $Adj(w) - v$, that has not been previously examined. This process is repeated until an element w in a row R_i is connected to an element in the next row R_{i+1}. Suppose *REROUT* backtracks to u and u is an element of the first row, then the next element to be chosen as the starting element for rerouting is the element in $(0, (col(u) + 1)$. And if *REROUT* fails for this case also, then the element in $(0, (col(u) - 1)$ is chosen to be the starting element for rerouting. Thus the starting element alternates between left and right of $(0, col(u))$. This process is repeated until a logical column is formed. *REROUT* will stop at the row R_m as at least the left boundary C_k is a feasible logical column. When *REROUT* terminate in this iteration, a new logical column based on C_k, denoted as C'_k, is obtained. We call C'_k the revised column of C_k.

In the next iteration, *REROUT* attempts to obtain the revised logical column for C_{k-1}. The left boundary $\mathcal{B}_l$ and the right boundary $\mathcal{B}_r$ are updated to C_{k-1} and the just revised column C'_k, respectively. The revision process terminates when all logical columns $C_1, C_2, \cdots, C_k$ have been revised. The resultant target

```
Boolean REROUT($H$, $S$, $curt$, $\boldsymbol{B}_l$, $\boldsymbol{B}_r$, $C'$);
/* revise the logical column $B_l$ from the element curt down to the last row $R_m$ */
begin
    repeat
        if there are unmarked elements in Adj(curt) then
        begin
            q := the first priority unmarked element in Adj(curt)
                and bounded by $\boldsymbol{B}_r$;
            pred(q) := curt; /* q is connected to curt */
            curt := q;
            mark q;
        end
        else curt := pred(curt); /* backtracking */
    until ((curt $\in R_m$)) or (curt $\in R_1$));
    if (curt $\in R_m$) then return true;
                    else return false;
end.

Algorithm RGCR($H$, $S$, $T$, $T'$);
/* revise the target array $T$ to $T'$. $T=\{C_1, C_2, \ldots, C_k\}$, $T'=\{C'_1, C'_2, \ldots, C'_k\}$. */
begin
    $\boldsymbol{B}_l := C_k$, $\boldsymbol{B}_r$ := nil;
    for i := k down to 1 do
    begin
        temp := the column number of the rightmost element in $\boldsymbol{B}_l$;
        curt := the element of column temp and row one;
        $C'$ := nil;
        While ($C'$ == nil) do
        begin
            if (REROUT($H$, $S$, curt, $\boldsymbol{B}_l$, $\boldsymbol{B}_r$, $C_i'$))
            then Store the revised column into $C'$;
            else curt := the closest element to curt (either on right
                        or left) which has not been examined;
        end
        $\boldsymbol{B}_l := C_{i\text{-}1}$, $\boldsymbol{B}_r := C'_i$;
    end;
end.
```


Solution of *GCR*, 24 irregular links

Solution of *RGCR*, 11 irregular links

The optimal solution, 8 irregular links

Fig. 2. The formal descriptions of *REROUT* and *RGCR*. The examples for the solutions of *GCR* and *RGCR* and the corresponding optimal solution for a 8 × 10 host array with 12 faults.

array, denoted as T', is called the revised target array of T. Fig. 2 shows the detailed description of the algorithm *RGCR*.

It is easy to see that, in rerouting, the selection of the highest priority element for inclusion into a logical column at each step is a greedy choice. Intuitively, it uses as many vertical links (low power interconnects) as possible to form the logical columns. Like *GCR*, it obviously runs in linear time, *i.e.*, the running time of the revision procedure is $O(N)$, *i.e.* the number of valid interconnections in the host array with N fault-free elements. In addition, there are also k independent logical columns in T' as each revised column C_i' is produced in the area bounded by C_i and C_{i+1}', $i = 1, 2, \cdots, k$. In other words, the revised algorithm does not decrease the size of the target array constructed by *GCR*. Hence, the harvest remains the same.

To summarize, Fig 2 shows the running results of the algorithms described in this section for a random instance of an 8 × 10 host array. The slant lines represent the irregular link-ways. The shaded boxes stand for faulty elements and the white ones stand for the fault-free elements. There are 8 irregular link-ways in the optimal solution for this instance. *GCR* produces a solution with 24 irregular link-ways while the proposed algorithm *RGCR* produces a nearly optimal solution with only 11 irregular link-ways.

4 Experimental Results

The proposed algorithms and *GCR* are implemented in C and run on a Pentium IV computer with one GB RAM. Both random fault model and clustered fault model are considered. These are acceptable because the random faults often occur during the system usage in real time configuration, while the clustered faults are typically caused during fabrication process.

The whole experiment is divided into two parts - one is for uniform fault distribution in the whole mesh, which is corresponding to random fault model. In this, the algorithms are run for different sized mesh. And, the other is for the uniform fault distribution in different localized portions of the mesh, which is corresponding to the clustered fault model. In this, the algorithms are run in 256 × 256 sized array for different fault density. Data are collected for different sized host arrays for three faulty density 0.1%, 1% and 10% of the size of the host array (same as in [11,12]), averaged over 10 random instances with the decimal being rounded off to the lower side in all cases.

Table 1. The performance comparison of the algorithms *GCR* and *RGCR* for uniform fault distribution, average of 10 random instances of different size

Host array		Target array	Performance		
Size	Fault	Size	*GCR*	*RGCR*	
$m \times n$	(%)	$m \times k$	*No.ir*	*No.ir*	*imp*
32 × 32	0.1	32 × 31	33	0	100%
32 × 32	1	32 × 30	265	76	71%
32 × 32	10	32 × 23	452	185	59%
64 × 64	0.1	64 × 63	292	59	80%
64 × 64	1	64 × 60	1471	403	73%
64 × 64	10	64 × 47	1992	885	56%
128 × 128	0.1	128 × 126	2012	499	75%
128 × 128	1	128 × 122	8168	3038	63%
128 × 128	10	128 × 95	8294	4138	50%
256 × 256	0.1	256 × 253	13827	3981	71%
256 × 256	1	256 × 246	38155	16947	56%
256 × 256	10	256 × 193	34167	18510	46%

In Table 1 and Table 2, the attribute *No.ir* denotes the number of the irregular link-ways. *imp* stands for the improvement in *No.ir* over *GCR*. It is calculated by

$$(1 - \frac{No.ir\_of\_RGCR}{No.ir\_of\_GCR}) \times 100\%.$$

In Table 1, data are collected for host arrays of different sizes from 32 × 32 to 256 × 256. For each algorithm, for smaller or medium host arrays, *No.ir* increases with the increase in the number of faulty elements in the host array. This is because that there will be a percentage faulty density beyond which *No.ir* will fall, as *No.ir* is equal to 0 both for 0% fault and for 100% fault. While for larger host arrays, *No.ir* first increases and then decreases due to heavy decrease in target size as the fault density increases beyond a certain point. For example, for *GCR*, *No.ir* is 33, 265 and 452 for size 32 × 32 and fault densities 0.1%, 1% and 10%, respectively. But for size 256 × 256 and fault densities 0.1%, 1% and 10%, *No.ir* increase to 13827, 38155 and 34167, respectively.

The improvement of *RGCR* over *GCR* is significant, especially for the host arrays of small size or small fault density, which is occurred more frequently in applications. For example, the improvements are 100% and 80% for the size 32 × 32 with 0.1% faulty elements and for the size 64 × 64 with the same faulty density, respectively.

Table 2. The performance comparison of the algorithms *GCR* and *RGCR* for clustered faulty distribution, average of 10 random instances of different size

Host array		Target array	Performance		
Size	Fault	Size	*GCR*	*RGCR*	
$m \times n$	(%)	$m \times k$	*No.ir*	*No.ir*	*imp*
1/8	0.1	256 × 249	3290	153	95%
	1	256 × 224	8446	0	100%
	10	NA	NA	NA	NA
1/4	0.1	256 × 251	5911	542	91%
	1	256 × 231	10179	842	92%
	10	NA	NA	NA	NA
1/2	0.1	256 × 252	10175	1620	84%
	1	256 × 241	16918	4553	73%
	10	256 × 145	15430	1016	93%
3/4	0.1	256 × 253	12553	2941	77%
	1	256 × 244	27090	10338	62%
	10	256 × 176	26208	8035	69%
whole	0.1	256 × 253	13827	3981	71%
	1	256 × 246	38155	16947	56%
	10	256 × 193	34167	18510	46%

In Table 2, data are collected for 256 × 256 sized host array, averaged over 10 random instances, each for a localized fault in the center 1/8, 1/4, 1/2, 3/4 and full portion of the host array. Here, center 1/8, 1/4, 1/2, 3/4 and *whole* in Table 2, is the situation when faulty elements are located in center area consisting of 32 ∗ 32 elements, 64 ∗ 64 elements, 128 ∗ 128 elements, 192 ∗ 192 elements and the full host array, respectively. For center 1/8 and 1/4, considering 10% fault density was not possible as 10% of 256 × 256 comes out to be 6554 elements, while center 1/8 comes out to be a 32 ∗ 32 mesh, *i.e.*, 1024 elements, and center

1/4 comes out to be 64 × 64 mesh, *i.e.*, 4096 elements. So it is clear that these two cannot accommodate 10% faulty elements. Hence, the row for these cases is marked as NA (Not Applicable).

From table 2, we observe that, for 1% faulty density and localized fault in center 1/8 portion of the host array, $No.ir$ comes out to be 0. This is because the center gets so much concentrated with the faulty elements that no logical column passes through that portion in which the faulty elements are located. *GCR* is not better for performance because it uses the leftmost strategy. The improvement over *GCR* is more significant for localized fault distribution than for random fault distribution. For example, for 1% fault density, the improvements of *RGCR* for center 1/8 and 1/4 fault distribution are 100% and 92%, respectively. While for the fault distribution in center 3/4 and whole host array, the improvements are 62% and 56%, respectively. In other words, the improvement increases as the spread of faulty elements in the host array decreases.

As can be seen from above tables, for a given percentage of faulty processors in the host array, as the size of the array increases, the number of irregular links in the target array increases. In all the cases, the revised algorithm gives much better results than the older *GCR* algorithm. Even for arrays of a specific size, as the percentage of faulty elements in the array increases, our new algorithm performs much better than the old algorithm. We can therefore say that for all cases our algorithms gives us a high performance solution that consumes much low power than the solutions of the older algorithms like *GCR*.

5 Conclusions

High performance VLSI design has emerged as a major theme in the electronics industry today. It can be achieved at all levels of the VLSI system. In this paper, we have proposed a new problem based on a high performance solution for reconfigurable VLSI arrays and have presented an algorithm for the same. The proposed algorithm is based on a heuristic strategy that is easy to implement. Its performance is very high and the harvest remains unchanged. Experimental results reflect the underlying characteristics of the proposed algorithm.

References

1. T. Leighton and A. E. Gamal. "Wafer-scal Integration of Systoric Arrays", *IEEE Trans. on Computer*, vol. 34, no. 5, pp. 448-461, May 1985.
2. C. W. H Lam, H. F. Li and R. Jakakumar, "A Study of Two Approaches for Reconfiguring Fault-tolerant Systoric Array", *IEEE Trans. on Computers*, vol. 38, no. 6, pp. 833-844, June 1989.
3. I. Koren and A. D. Singh, "Fault Tolerance in VLSI Circuits", *Computer*, vol. 23, no. 7, pp. 73-83, July 1990.
4. Y. Y. Chen, S. J. Upadhyaya and C. H. Cheng, "A Comprehensive Reconfiguration Scheme for Fault-tolerant VLSI/WSI Array Processors", *IEEE Trans. on Computers*, vol. 46, no. 12, pp. 1363-1371, Dec. 1997.

5. T. Horita and I. Takanami, “Fault-tolerant Processor Arrays Based on the 1.5-track Switches with Flexible Spare Distributions”, *IEEE Trans. on Computers*, vol. 49, no. 6, pp. 542-552, June 2000.
6. S. Y. Kuo and W. K. Fuchs, “Efficient Spare Allocation for Reconfigurable Arrays”, *IEEE Design and Test*, vol. 4, no. 7, pp. 24-31, Feb. 1987.
7. C. L. Wey and F. Lombardi, “On the Repair of Redundant RAM’s”, *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 6, no. 2, pp. 222-231, Mar. 1987.
8. Li Zhang, “Fault-Tolerant meshes with small degree”, *IEEE Trans. on Computers*, col. 51, No.5, pp.553-560, May, 2002.
9. S. Y. Kuo and I. Y. Chen, “Efficient reconfiguration algorithms for degradable VLSI/WSI arrays,” *IEEE Trans. Computer-Aided Design*, vol. 11, no. 10, pp. 1289-1300, Oct. 1992.
10. C. P. Low and H. W. Leong, “On the reconfiguration of degradable VLSI/WSI arrays,” *IEEE Trans. Computer-Aided Design of integrated circuits and systems*, vol. 16, no. 10, pp. 1213-1221, Oct. 1997.
11. C. P. Low, “An efficient reconfiguration algorithm for degradable VLSI/WSI arrays,” *IEEE Trans. on Computers*, vol. 49, no. 6, pp.553-559, June 2000.
12. Wu Jigang, Schroder Heiko & Srikanthan Thambipillai, “New architecture and algorithms for degradable VLSI/WSI arrays”, in Proc. Of 8th International Computing and Combinatorics Conference, 2002, Singapore (COCOON’O2), *Lecture Notes in Computer Science*, vol. 2387, pp.181-190, Aug, 2002.
13. A. V. Aho, J. E. Hopcroft, and J. D. Ullman, “The design and analysis of computer algorithms”, Addison-Wesley, Reading, Mass., 1974.

Order Independent Transparency for Image Composition Parallel Rendering Machines

Woo-Chan Park[1], Tack-Don Han[2], and Sung-Bong Yang[2]

[1] Department of Internet Engineering,
Sejong University, Seoul 143-747, Korea,
pwchan@sejong.ac.kr
[2] Department of Computer Science,
Yonsei University, Seoul 120-749 Korea,
{hantack}@kurene.yonsei.ac.kr
{yang}@cs.yonsei.ac.kr

Abstract. In this paper, a hybrid architecture composed of both the object-order and the image-order rendering engines is proposed to achieve the order independent transparency on the image composition architecture. The proposed architecture utilizes the features of the object-order which may provide high performance and the image-order which can obtain the depth order of all primitives from a viewpoint for a given pixel. We will discuss a scalable architecture for image order rendering engines to improve the processing capability of the transparent primitives, a load distribution technique for hardware efficiency, and a preliminary timing analysis.

1 Introduction

3D computer graphics is a core field of study in developing multi media computing environment. In order to support realistic scene using 3D computer graphics, a special purpose high performance 3D accelerator is required. Recently, low cost and high performance 3D graphics accelerators are adopted rapidly in PCs and game machines[1,2,3].

To generate high-quality images, solid models composed of more than several millions polygons are often used. To display such a model at a rate of 30 frames per second, more than one hundred million polygons must be processed in one second. To achieve this goal in current technology, several tens of the graphic processors are required. Thus, parallel rendering using many graphics processors is an essential research issue.

According to [4], graphics machine architectures can be categorized into the three types: *sort-first architecture*, *sort-middle architecture*, *sort-last architecture*. Among them, sort-last architecture is a scalable architecture because the required bandwidth of its communication network is almost constant against the number of polygons. Thus, sort-last architecture is quite suitable for a large-scale rendering system.

P.-C. Yew and J. Xue (Eds.): ACSAC 2004, LNCS 3189, pp. 449–460, 2004.

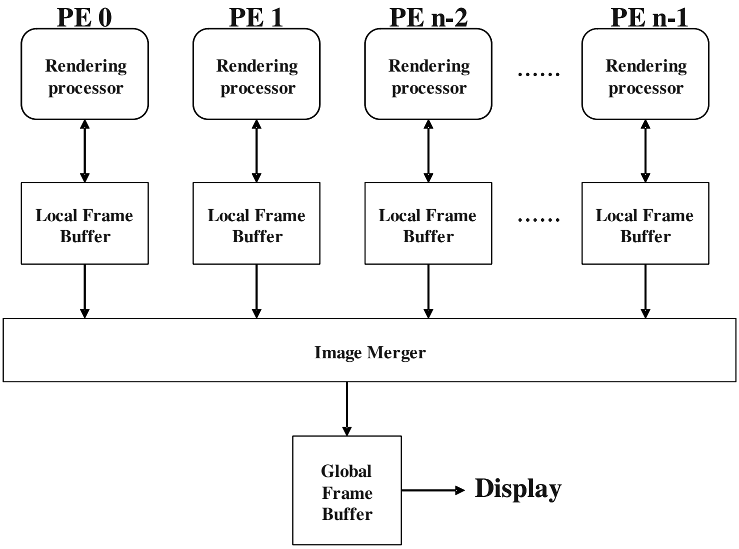


Fig. 1. An image composition architecture

One of the typical sort-last architectures is an image composition architecture [5,6,7,8,9]. Figure 1 shows the overall structure of image composition architecture. All polygonal model data are distributed into each rendering processor which generates a subimage with its own frame buffer, called a local frame buffer. The contents of all the local frame buffers are merged periodically by the image merger. During image merging, the depth comparisons with the contents of the same screen address for each local frame buffer should be performed to accomplish hidden surface removal. The final merged image is then transmitted into the global frame buffer.

In a realistic 3D scene, both opaque and transparent primitives are mixed each other. To generate a rendered final image properly with the transparent primitives, the order dependent transparency problem must be solved. That is, the processing order depends on the depth order by a viewpoint, not on the input order. Order dependent transparency problem may cause the serious performance degradation as the number of transparent primitives increases rapidly. Therefore, the order independent transparency, the opposite concept of the order dependent transparency, is a major for providing high performance rendering systems. But up until now we have not found any parallel 3D rendering machine supporting hardware accelerated order independent transparency.

In this paper, we propose a new method of the hardware accelerated order independent transparency for image composition in the parallel rendering architecture. To achieve this goal, we suggest a hybrid architecture composed of both the object order and the image order rendering engines. The proposed mechanism utilizes the features of the object order which may provide high performance and the image order which can obtain the depth order of all primitives from a viewpoint for a given pixel.

The proposed architecture has n object order rendering engines, from PE 0 to PE n-1 as in Figure 1 and one image order rendering engine, PE n, where n is the number of PEs. All the opaque primitives are allocated with each object order rendering engine. All the transparent primitives are sent to the image order rendering engine. Thus subimages generated from PE 0 to PE n-1 are merged into a single image during image merging. This merged image is fed into the image order rendering engine, PE n. With this single merged image and the information of the transparent primitives, PE n calculates the final image with order independent transparency. We also provide a scalable architecture for the image order rendering engines to improve the processing capability of transparent primitives and load distribution technique for hardware efficiency.

In the next section, we present a brief overview of the object order rendering, the image order rendering, the order independent transparency. Section 3 illustrates the proposed image composition architecture and its pipeline flow. We describe how the proposed architecture handles the order independent transparency. The timing analyses of the image merger and the image order rendering engine, and the scalability are also discussed. Section 4 concludes this paper with future research.

2 Background

In this section, we present a brief overview of the object order rendering and the image order rendering methods. We also discuss the order independent transparency.

2.1 Object Order and Image Order Rendering Methods

Polygonal rendering algorithms can be classified into the object order rendering and the image order rendering according to the rendering processing order for the input primitives [10]. In the object order rendering the processing at the rasterization step is performed primitive by primitive, while in the image order it is done pixel by pixel. According to their features, an object order rendering system is suitable for high performance systems and current design approaches [1,2,3,5,9], while the image order is low cost systems and later approaches [11,12,13,14]. In most of parallel rendering machines, an object order rendering engine is adopted for each rendering processor for high performance computation.

The rendering process consists of geometry processing, rasterization, and display refresh steps. In an object order rendering system, the processing order at the rasterization step is performed primitive by primitive. Thus the geometry processing and the rasterization steps are pipelined between primitives and the object order rendering and the display refresh steps are pipelined between frames. An object order rendering system must have a full-screen sized frame buffer, consisted of a depth buffer and a color buffer, for hidden surface removal operations. To overlap the executions of the rendering and the screen refresh steps, double buffering for the frame buffer are used. The front frame buffer denotes the frame buffer used in the rendering step and the back frame buffer is used by the screen refresh step. Figure 2 shows the pipeline flow of the object order rendering system between two frames.

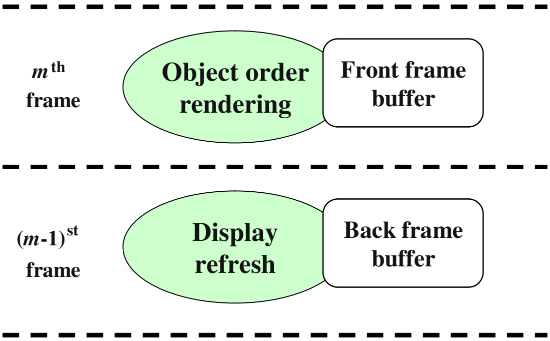


Fig. 2. The pipeline flow of the object order rendering method between two frames

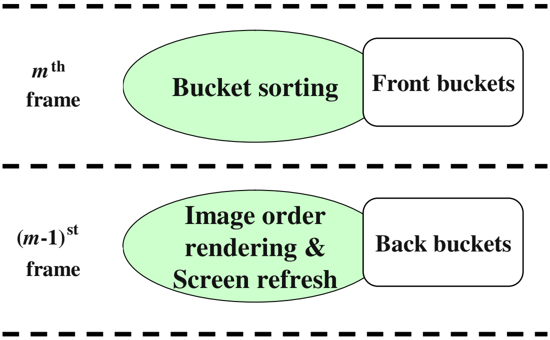

m^{th} frame
Bucket sorting
Front buckets
$(m\text{-}1)^{\text{st}}$ frame
Image order rendering & Screen refresh
Back buckets

Fig. 3. The pipeline flow of the image order rendering method between two frames

In an image order rendering system, the processing order at the rasterization step is performed pixel by pixel. For example, if the screen address begins from (0, 0) and ends with (v, w), the rasterization step is accomplished from (0, 0) to (v, w). In rendering pixel by pixel, the lists of the all primitives, after geometry transformation, overlaying with a dedicated pixel must be kept for each pixel. These lists are called the *buckets* and bucket sorting denotes this listing. Thus the bucket sorting step including geometry processing and the image order rendering step including screen refresh are pipelined between frames. Figure 3 shows the pipeline flow of the image order rendering system between two frames.

The scan-line algorithm is a typical method of the image order rendering [10]. All primitives transformed into the screen space are assigned into buckets provided per scan-line. To avoid considering primitives that do not contribute to the current scan-line, the scan-line algorithm requires primitives to be transformed into the screen space and to sort the buckets according to the first scan-line in which they appear. After bucket sorting, rendering is performed with respect to each scan-line. Because the finally rendered data are generated by scan-line order, screen refresh can be performed immediately after scan-line rendering. Thus no frame buffer is needed and only a small amount of buffering is required. In this paper, the scan-line algorithm is used in the image order rendering engine.

2.2 Order Independent Transparency

In a realistic 3D scene, both opaque and transparent primitives are mixed each other. To generate a final rendered image properly, the order dependent transparency problem should be solved. Figure 4 shows an example of order dependent transparency with three fragments for a given pixel.

In Figure 4, A is a yellow transparent fragment, B is a blue opaque fragment, and C is a red opaque fragment. We assume that A is followed by B and B is followed by C in the point of depth value for the current view point. Then the final color results in green which is generated by blending the yellow color of A and the blue color of B. However, if the processing order of these three fragments are C, A, and B, then a wrong color will be generated. That is, when C comes first, the current color is red. When A comes next, the current color becomes orange and the depth value of the current color is that of A. When B comes last, B is ignored because the current depth value is smaller then depth value of B. Therefore, the calculated final color is orange, which is wrong. Therefore, the processing orders should be the same as the depth order to achieve the correct final color.

The order dependent transparency problem may cause serious performance degradation as the number of transparent primitives increases rapidly. Therefore, the order independent transparency is a crucial issue for high performance rendering systems. But, up until now we have not found any high performance 3D rendering machine supporting hardware accelerated.

Several order independent transparency techniques for object order rendering based on the A-buffer algorithm have been proposed [15,16,17,18,19]. But, these algorithms require either, for each pixel, the infinite number of lists of all the primitives overlaying with the dedicated pixel or multiple passes which are not suitable for high performance. On the other hand, image order rendering

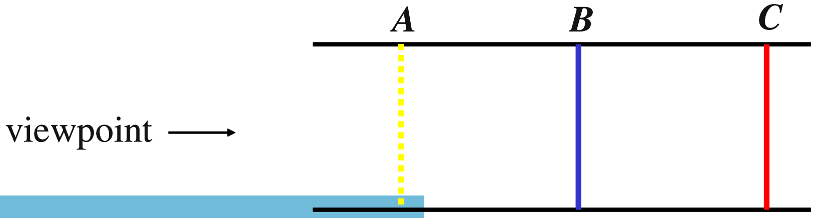


Fig. 4. An example of order independent transparency

techniques to support order independent transparency based on the A-buffer algorithm have been proposed in [13,14]. However, high performance rendering cannot be achieved with image order rendering technique.

3 Proposed Image Composition Architecture

In this section, an image composition architecture supporting hardware accelerated order independent transparency is proposed. We discuss its execution flow, preliminary timing analysis, and scalability.

3.1 Proposed Image Composition Architecture

Figure 5 shows the block diagram of the proposed image composition architecture, which can be divided into rendering accelerator, frame buffer, and display subsystems. The proposed architecture is capable of supporting hardware accelerated order independent transparency. To achieve this goal, hybrid architecture made up of the object order and the image order rendering engines are provided. The proposed mechanism utilizes the advantages of both the object order rendering and the image order rendering methods.

The proposed architecture has n object order rendering engine(ORE)s from PE 0 to PE n-1, and PE n with an image order rendering engine(IRE). Each PE consists of geometry engine(GE), either ORE or IRE, and a local memory which

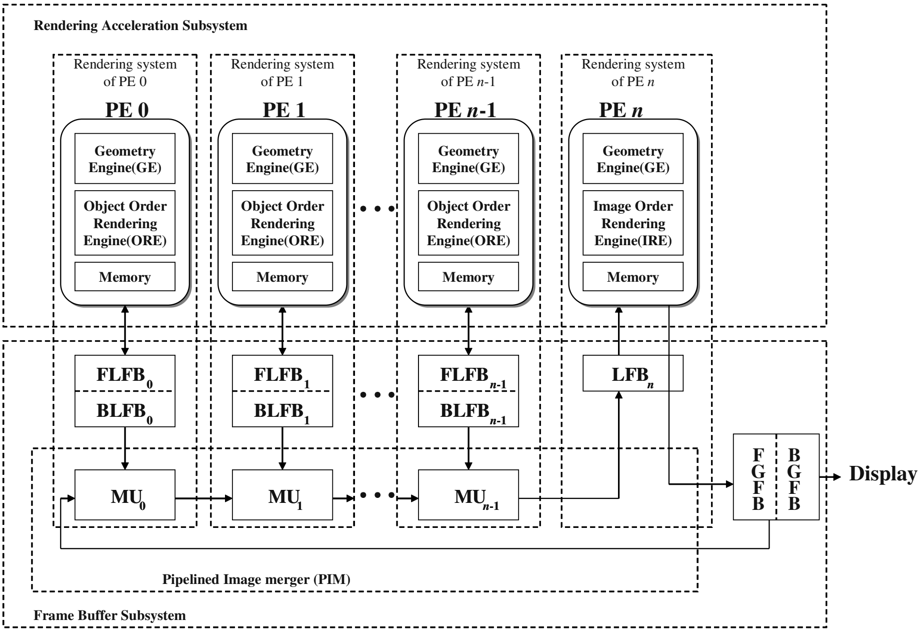


Fig. 5. Block diagram of the proposed image composition architecture

is not a frame memory but a working memory for the PE. The rendering system from PE 0 to PE n-1 is identical to that of the conventional sort-last architecture, while PE n is provided to achieve OIT. The local frame buffer(LFB) of each ORE is double-buffered so that the OREs can generate the image of the next frame in parallel with the image composition of the previous frame. One buffer is called the front local frame buffer(FLFB) and the other is called the back local frame buffer(BLFB). This double buffering scheme is also used in the buckets of IRE and in global frame buffer(GFB).

In the proposed architecture, the rendering systems from PE 0 to PE n-1 perform rendering all opaque primitives with an object order base and PE n performs rendering all transparent primitives with an image order base. Subimages generated from PE 0 to PE n-1 are merged into a single image which is transmitted into LFB_n. With this single merged image and the information of the transparent primitives, IRE calculates the final image with order independent transparency.

Pipelined image merger(PIM), provided in [6], is made up of a linearly connected array of n merging unit(MU)s. It performs image merging with n BLFBs and transmits the final merged image into GFB in a pipelined fashion. Each MU receives two pixel values and outputs the one with the smaller screen depth. We let the screen address begins from (0, 0) and ends with (v, w). BLFB_0 denotes the BLFB of PE 0, BLFB_1 is the BLFB of PE 1, and so on. MU_0 denotes the MU connected with PE 0, MU_1 is the MU connected with PE 1, and so on.

The execution behavior of PIM can be described as follows. In the first cycle, MU_0 performs depth comparison with (0, 0)'s color and the depth data of BLFB_0 and (0, 0)'s color and depth data of FGFB, respectively, and transmits the results of color and depth data into MU_1. In the next cycle, MU_0 performs depth comparison with (0, 1)'s color and depth data of BLFB_0 and (0, 1)'s color and depth data of FGFB, and transmits the results of color and depth data into MU_1. Simultaneously, MU_1 performs depth comparison with (0, 0)'s color and depth data of BLFB_1 and the result values fed into the previous cycle, and transmits the results of color and depth data into MU_2. As PIM executes in this pipelined fashion, the final color data can be transmitted into FGFB.

3.2 Execution Flow of the Proposed Image Composition Architecture

All opaque primitives are allocated to each ORE and all transparent primitives are sent to IRE. A primitive allocation technique for load balance on the opaque primitives has been provided in [6]. In the first stage, from PE 0 to PE n-1, object order rendering is performed for the dedicated opaque primitives with geometry processing and rasterization steps. Simultaneously, PE n executes the bucket sorting for image order rendering through geometry processing with the transparent primitives. As a processing result, the rendered subimages for the opaque primitives are stored from FLFB_0 to FLFB_{n-1} and the bucket sorted result for the transparent primitives are stored in buckets, which reside in the local memory of PE n.

In the next step, the subimages stored from BLFB$_0$ to BLFB$_{n-1}$ are merged by PIM according to the raster scan order, from (0, 0) to (v, w), and the final merged image is transmitted into LFB$_n$ with a pipelined fashion. Simultaneously, IRE performs image order rendering with LFB$_n$, which hold the rendered result of all opaque primitives and the bucket sorted results of all transparent primitives. To perform simultaneously both the write operation from MU$_{n-1}$ and the read operation from IRE for two different memory addresses, a two-port memory should be used in LFB$_n$.

Using the scan-line algorithm for image order rendering, IRE should check all the color and depth values of the current scan-line of LFB$_n$. Therefore, they should be transmitted completely from MU$_{n-1}$ before performing rendering operation for the current scan-line. Thus, IRE cannot perform rendering operation until all color and depth values of the 0^{th} scan-line in LFB$_n$ are transmitted completely from MU$_{n-1}$. But the transfer time between MU$_{n-1}$ and LFB$_n$ is too short to affect overall performance, as shown in Section 3.5. The final rendering results of IRE are generated and transmitted into GFB scan-line by scan-line order. Finally, GFB performs the screen refresh operation.

3.3 Pipeline Flow of the Proposed Image Composition Architecture

Figure 6 shows a pipelined execution of the proposed image composition architecture with respect to three frames. If the current processing frame is m, object order rendering is performed from PE 0 to PE n-1 with all the opaque primitives and PE n executes the bucket sorting for image order rendering with all the transparent primitives. The rendering results for the opaque primitives are stored from FLFB$_0$ to FLFB$_{n-1}$ and the bucket sorted results for the transparent primitives are stored in front buckets. Simultaneously, for the (m-1)st frame, image merging from BLFB$_0$ to BLFB$_{n-1}$ is performed by PIM according to the raster scan order, and the final merged image is transmitted into LFB$_n$ with a pipelined fashion. Simultaneously, IRE of PE n performs image order rendering with LFB$_n$ and the back bucket. The rendering results of IRE are generated and transmitted into GFB scan-line by scan-line. For the (m-2)nd frame, screen refresh is performed with a final merged image in FGFB.

3.4 Example of the Order Independent Transparency

Figure 7 shows an example of the order independent transparency for the proposed image composition architecture and illustrates the input and output values of OREs and IRE for a given pixel. For the current viewpoint, the depth values of A, B, C, D, E, F, and G are in increasing order, i.e., A is the nearest primitive and G is the farthest primitive. We assume that A is yellow and transparent, B is blue and opaque, C is red and transparent, D is blue and opaque, E is yellow and opaque, F is black and opaque, and G is green and transparent.

Among the opaque primitives (B, D, E, and F) generated in OREs, B is the final depth result of depth comparison. With the transparent primitives (A, C,

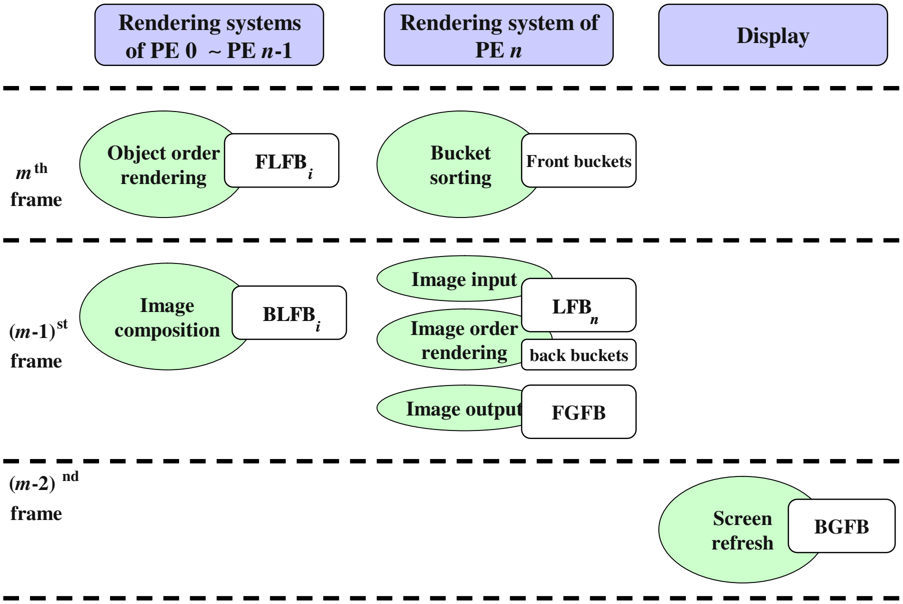


Fig. 6. Pipelined execution of the proposed architecture

and G) and B, image order rendering is performed in IRE. Therefore, green is generated as the final color with order independent transparency.

3.5 Timing Analysis of PIM and Image Order Renderer

In [6] each MU performs depth comparison and data selection through a 4-stage pipeline. Therefore, PIM, as a whole, constructs a $4n$-stage pipeline, where n represents the number of PEs. The time, T_m, needed for merging one full-screen image is equal to $(v + \beta) \cdot w \cdot t + 4 \cdot n \cdot t$, where v and w are horizontal and vertical screen sizes, respectively, t is the PIM clock period, and β is the number of PIM clocks required for overhead processing per scan-line in the GFB unit. The first term represents the time for scanning the entire screen and the second term represents the pipeline delay time. In [6] those parameters are v = 640, w = 480, n = 16, β = 20, and t = 80 nsec. Therefore, T_m = 25.3 msec, which is shorter than the target frame interval(33.3 msec).

Because the current screen resolution exceeds several times the resolution considered in [6] and the clock period of PIM can be shortened due to the advances in the semiconductor and network technologies, those parameters are not realistic for the current graphics environment and semiconductor technology. These parameters will be fixed after developing a prototype by the future work. However, considering the current technology, the parameters can be estimated reasonably as v = 1280, w = 1024, and t = 20 nsec, where n and β are not considered because the effect of those parameters is ignorable. Therefore, T_m = 26.7 msec, which is still sufficient time to support 30 frames per second.

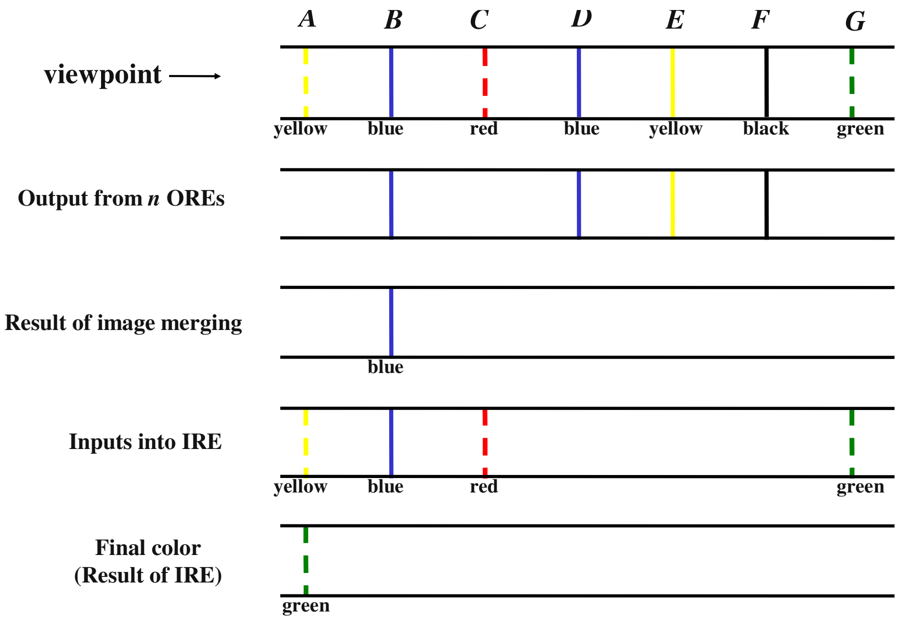


Fig. 7. An example of the order independent transparency on the proposed architecture

IRE cannot perform rendering operation until all color and depth values of the 0^{th} scan-line in LFB_n are transmitted completely from MU_{n-1}. T_{in} denotes the time taken for the transmission. T_{out} denote the time needed to transmit the rendered results of the final scan-line into GFB. T_{in} and T_{out} can be estimated as $(v + \beta) \cdot w \cdot t + 4 \cdot n \cdot t$ and $(v + \beta) \cdot w \cdot t$, respectively. Then, the actual time to perform the image order rendering at IRE is 33.3msec$-(T_{in} + T_{out})$. With $n = 16$, $T_{in} = 0.058$msec and $T_{out} = 0.053$msec in the case of [6] and $T_{in} = 0.038$msec and $T_{out} = 0.026$msec in the case of the estimated parameters for the current technology. Therefore, the performance degradation due to T_{in} and T_{out} with $n = 16$ is about $0.4 \sim 0.2\%$, which is negligible. Moreover, with n=1024, T_{in}=0.38msec and $T_{out} = 0.053$msec in the case of [6] and $T_{in} = 0.083$msec and $T_{out} = 0.026$msec in the case of the estimated parameters. Therefore, the performance degradation due to T_{in} and T_{out} with $n = 1024$ is about $1.3 \sim 0.4\%$, which is also negligible.

3.6 Scalability on the IRE

In Figure 5 only one IRE is used. Thus, when the number of the transparent primitives is so large that all transparent primitives cannot be processed within the target frame interval, the bottleneck point is the performance of IRE. Figure 8 shows the proposed image composition architecture with r IREs. To achieve scalable parallel processing for IRE, the per-scan-line parallelism is used for the scan-line algorithm.

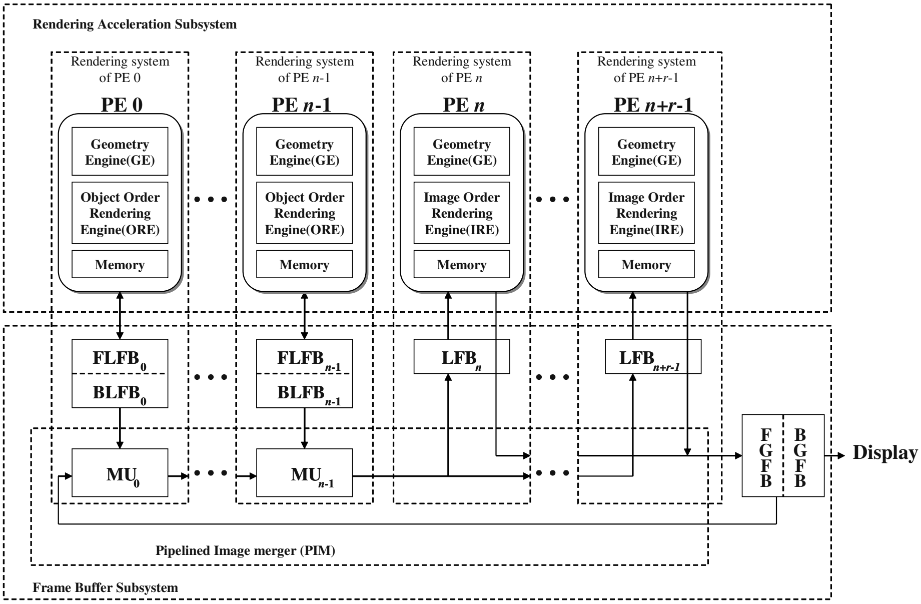


Fig. 8. Block diagram of proposed architecture with scalability on the IRE

In case that the full-screen consists of $r \cdot k$ scan-lines, each IRE has a copy of all transparent primitives and performs bucket sorting for all transparent primitives with dedicated k buckets instead of $r \cdot k$ buckets. Then, each IRE executes the image order rendering with an interleaving fashion. That is, IRE of PE n performs scan-line rendering with the 0^{th} scan-line, the r^{th} scan-line, and so on. IRE of PE $n+1$ performs scan-line rendering with the 1^{st} scan-line, the $(r+1)^{st}$ scan-line, and so on. By this sequence, all scan-lines can be allocated to r IREs. Simultaneously, subimages stored in BLFB_0 to BLFB_{n-1} are merged by PIM according to the raster scan order. Then, the merged image is transmitted into each LFB of IRE with an interleaving fashion. Therefore, overall performance of the image order rendering can be achieved with scalability. Finally, the rendered result of r IREs are also transmitted into GFB with an interleaving fashion.

4 Conclusion

In this paper, the order independent transparency problem for image composition in parallel rendering machines has been resolved by using hybrid architecture composed of both the object order rendering and the image order rendering engines. The proposed architecture is a scalable one with respect to both the object order rendering and the image order rendering engines.

References

1. M. Oka and M. Suzuoki. Designing and programming the emotion engine. IEEE Micro, 19(6):20–28, Nov. 1999.
2. A. K. Khan et al. A 150-MHz graphics rendering processor with 256-Mb embedded DRAM. IEEE Journal of Solid-State Circuits, 36(11):1775–1783, Nov. 2001.
3. Timo Aila, Ville Miettinen, and Petri Nordlund. Delay streams for graphics hardware. In Proceedings of SIGGRAPH, pages 792–800, 2003.
4. S. Molnar, M. Cox, M. Ellsworth, and H. Fuchs. A sorting classification of parallel rendering. IEEE Computer Graphics and Applications, 14(4):23–32, July 1994.
5. T. Ikedo and J. Ma. The Truga001: A scalable rendering processor. IEEE computer and graphics and applications, 18(2):59–79, March 1998.
6. S. Nishimura and T. Kunii. VC-1: A scalable graphics computer with virtual local frame buffers. In Proceedings of SIGGRAPH, pages 365–372, Aug. 1996.
7. S. Molnar, J. Eyles, and J. Poulton. PixelFlow: High-speed rendering using image composition. In Proceedings of SIGGRAPH, pages 231–240, July 1992.
8. J. Eyles, S. Molnar, J. Poulton, T. Greer, A. Lastra, N. England, and L. Westover. PixelFlow: The realization. In Proceedings of SIGGRAPH/Eurographics Workshop on graphics hardware, pp. 57–68, Aug. 1997.
9. M. Deering and D. Naegle. The SAGE Architecture. In Proceeddings of SIGGRAPH 2002, pages 683–692, July 2002.
10. J. D. Foley, A. Dam, S. K. Feiner, and J. F. Hughes. Computer Graphics, Principles and Practice. Second Edition, Addison-Wesley, Massachusetts, 1990.
11. J. Torborg and J. T. Kajiya. Talisman: Commodity Realtime 3D graphics for the PC. In Proceedings of SIGGRAPH, pages 353–363, 1996
12. M. Deering, S. Winner, B. Schediwy, C. Duffy, and N. Hunt. The triangle processor and normal vector shader: A VLSI system for high performance graphics. In Proceedings of SIGGRAPH, pages 21–30, 1988.
13. M. Kelley, S. Winner, and K. Gould. A scalable hardware render accelerator using a modified scanline algorithm. In Proceedings of SIGGRAPH, pages 241–248, 1992.
14. M. Kelley, K. Gould, B. Pease, S. Winner, and A. Yen. Hardware accelerated rendering of CSG and transparency. In proceedings of SIGGRAPH, pages 177–184, 1994.
15. L. Carpenter. The A-buffer, and antialiased hidden surface method. In Proceedings of SIGGRAPH, pages 103–108, 1984.
16. S. Winner, M. Kelly, B. Pease, B. Rivard, and A. Yen. Hardware accelerated rendering of antialiasing using a modified A-buffer algorithm. In Proceedings of SIGGRAPH, pages 307–316, 1997.
17. A. Mammeb. Transparency and antialiasing algorithms implemented with virtual pixel maps technique. IEEE computer and graphics and applications, 9(4):43–55, July 1989.
18. C. M. Wittenbrink. R-Buffer: A pointerless A-buffer hardware architecture. In Proceedings of SIGGRAPH/Eurographics Workshop on graphics hardware, pages 73–80, 2001.
19. J. A. Lee and L. S. Kim. Single-pass full-screen hardware accelerated anti-aliasing. In Proceedings of SIGGRAPH/Eurographics Workshop on graphics hardware, pages 67–75, 2000.

An Authorization Architecture Oriented to Engineering and Scientific Computation in Grid Environments

Changqin Huang [1,2], Guanghua Song[1, 2], Yao Zheng[1,2], and Deren Chen[1]

[1] College of Computer Science, Zhejiang University, Hangzhou, 310027, P. R. China
[2] Center for Engineering and Scientific Computation, Zhejiang University,
Hangzhou, 310027, P. R. China
{cqhuang, ghsong, yao.zheng, drchen}@zju.edu.cn

Abstract. Large-scale scientific and engineering computation is normally accomplished through the interaction of collaborating groups and diverse heterogeneous resources. Grid computing is emerging as an applicable paradigm, whilst, there is a critical challenge of authorization in the grid infrastructure. This paper proposes a Parallelized Subtask-level Authorization Service architecture (PSAS) based on the least privilege principle, and presents a context-aware authorization approach and a flexible task management mechanism. The minimization of the privileges is conducted by decomposing the parallelizable task and re-allotting the privileges required for each subtask. The dynamic authorization is carried out by constructing a multi-value community policy and adaptively transiting the mapping. Besides applying a relevant management policy, a delegation mechanism collaboratively performs the authorization delegation for task management. In the enforcement mechanisms involved, the authors have extended the RSL specification and the proxy certificate, and have modified the Globus gatekeeper, jobmanager and the GASS library to allow authorization callouts. Therefore the authorization requirement of an application is effectively met in the presented architecture.

1 Introduction

Grid Computing [1] emerges as a promising paradigm for coordinating the sharing of computational and data resource and wide-area distributed computing across organizational boundaries. The sharing of code and data on the grid gives rise to many great challenges. Grid infrastructure software such as Legion [2] and Globus [3] enables a user to identify and use the best available resource(s) irrespective of resource location and ownership. However, realizing such a pervasive grid infrastructure presents many challenges due to its inherent heterogeneity, multi-domain characteristic, and highly dynamic nature. One critical challenge is providing authentication, authorization and access control guarantees.

Among relevant grid applications, due to the capability of full utilization of many valuable resources, engineering and scientific computing is suited for being solved in grid environments. This type of task is commonly either computation-intensive or

P.-C. Yew and J. Xue (Eds.): ACSAC 2004, LNCS 3189, pp. 461–472, 2004.

data-intensive, the problem granularity is widely large and computational tasks are often long-lived. It needs be divided into many subtasks, and then be distributed to many relevant nodes and run in parallel, and the management of subtasks is dynamic. The issue needs not only fine-grained authorization for resource usage and management but also fine-grained authorization for task management to meet the needs of this type of application.

In this paper, we focus on the security requirements posed by engineering and scientific computation applications in grid. We present the Parallelized Subtask-Level Service Authorization (PSAS) architecture for fine-grained authorization policies and enforcement mechanism for both resource usage/management and task management. The context-aware authorization is exercised by mapping a community member to a multi-value community policy and adaptive transition, and a delegation mechanism collaboratively performs task management together with a relevant management policy. It enforces these mechanisms to enable fine-grained authorization based on Globus Toolkit version 2.2.

This paper is organized as follows: Section 2 reviews background and related work in the arena of grid security. In section 3, the proposed authorization architecture and overall policy are described. Context-aware authorization is presented in Section 4. Section 5 describes the current implementation of the architecture within Globus. Finally, conclusions and future work are addressed in Section 6.

2 Backgrounds and Related Work

2.1 Authorization in Grid Middleware

As the rapid advancement of the grid researches and applications, diverse grid middlewares are widely developed and deployed. At present, there are three main pieces of grid middlewares, Globus [3], Legion [2], and UNICORE [17]. The Globus toolkit is the most popular grid environment and the de facto grid standard. However its current security services are yet poor, for example: use of static user accounts, coarse granularity, and application dependent enforcement mechanisms. Globus has adopted the Grid Security Infrastructure (GSI) [5] as the primary authentication mechanism. GSI defines single sign-on algorithms and protocols, cross-domain authentication protocols, and temporary credentials called proxy credentials to support hierarchical delegation [6]. Main weaknesses of the Globus security services are described as follows:

1. Globus deals with all privileges of subtask irrespective of the privilege difference among its subtasks. That is, after the simple authentication is exercised, the resource allows the task to use all privileges of the user; similarly, so do subtasks run in parallel. It violates commonly the least privilege principle [7].
2. The issues of the context-aware community policy are not concentrated on, so the authorization of resource usage and task management is not flexibly characterized by the community. The scenario is not suited for large-scale wide-area collaboratively scientific computation in Virtual organization.

3. In Globus, normally, task management is only the responsibility of the users who have submitted the job. Due to the dynamic environment and the long-lived feature of engineering and scientific computation, this coarse-grain authorization for task management cannot meet the need of agile job management in VO.

2.2 Related Work

In recent years, many grid security issues (architectures, policies and enforcement mechanisms, etc) have been researched. And the related researches are making great progress. Among many related works, main researches are presented in the following:

I. Foster et al. [5] provide the basis of current grid security: "grid-map" mechanism, mapping grid entities to local user accounts at the grid resources, is a common approach to authorization. A grid request is allowed if such a mapping exists and the request will be served with all the privileges configured for the local user account. Obviously, these authorization and access control mechanisms are not suitable for flexible authorization decision.

L. Pearlman et al. [8] propose the Community Authorization Service (CAS) architecture. Based on CAS, resource providers grant access to a community accounts as a whole, and community administrators then decide what subset of a community's rights an individual member will have. Drawbacks of this approach include that enforcement mechanism does not support the use of legacy application, that the approach of limiting the group's privileges violates the least-privilege principle and that it does not consider authorization issue of task management.

W. Johnston et al. [9] provide grid resources and resource administrators with distributed mechanisms to define resource usage policy by multiple stakeholders and make dynamic authorization decisions based on supplied credentials and applicable usage policy statements. This system binds user attributes and privileges through attribute certificates (ACs) and thus separates authentication from authorization. Fine-grained access decisions are enforced via such policies and user attributes. However, It does not provide convenience for the use of legacy applications, and does not consider authorization issue of task management.

R. Alfieri et al. [10] present a system conceptually similar to CAS: the Virtual Organization Membership Service (VOMS), which also has a community centric attribute server that issues authorization attributes to members of the community. M. Lorch et al. [11] give the same architecture, called PRIMA. Except that in PRIMA the attributes are not issued by a community server but rather come directly from the individual attribute authorities, PRIMA and VOMS have similar security mechanisms. They utilize expressive enforcement mechanisms and/or dynamic account to facilitate highly dynamic authorization policies and least privilege access to resources. However, they do not consider authorization issue of task management, and in their study, the overhead of authorization management is larger. They only support the creation of small, transient and ad hoc communities.

Besides the typical paradigms mentioned above, M. Lorch et al. [12] enable the high-level management of such fine grained privileges based on PKIX attribute certificates and enforce resulting access policies through readily available POSIX operating system extensions. Although it enables partly the secure execution of legacy applications, it is mainly oriented to collaborating computing scenarios for small, ad hoc working groups. G. Zhang et al. [13] present the SESAME dynamic context-aware access control mechanism for pervasive Grid applications by extending the classic role based access control (RBAC) [14]. SESAME complements current authorization mechanisms to dynamically grant and adapt permissions to users based on their current context. But, monitoring grid context in time is high-cost. K. Keahey et al. [15] describe the design and implementation of an authorization system allowing for enforcement of fine-grained policies and VO-wide management of remote jobs. However, it does not specify and enforce community policies for resource usage and management currently. S. Kim et al. [16] give a WAS architecture to support a restricted proxy credential and rights management by using workflow. It does not consider the actual conditions of large-scale task running at many nodes in parallel, the large overhead of fine-grained division of task and associated authorization confine its application to a limited area.

3 PSAS Architecture

3.1 PSAS Architecture Overview

PSAS architecture is concerned with the different privilege requirements of subtasks, user privilege of resource usage and resource policy in virtual community, task management policy and task management delegation. So PSAS architecture includes three functional modules and a shared enforcement mechanism as shown in Figure 1.

To minimize privileges of a task, the parallelizable task is decomposed and the least privileges required for each subtask is re-allotted after analyzing the source codes of the task. This contribution is described in the part of Subtask-level authorization module in the next sub-section. To apply a flexible task management, a delegation mechanism collaboratively performs the authorization delegation for task management together with a relevant management policy. Its details exist in the part of Task management authorization module in the next sub-section. A context-aware authorization approach is another contribution based on PSAS architecture, and it is presented in Section 4.

3.2 Privilege Management and Overall Authorization Policy

Besides the shared enforcement mechanism, in PSAS, there exist three modules to implement Privilege management and overall authorization policy: Subtask-level authorization module, Community authorization module, and Task management authorization module.

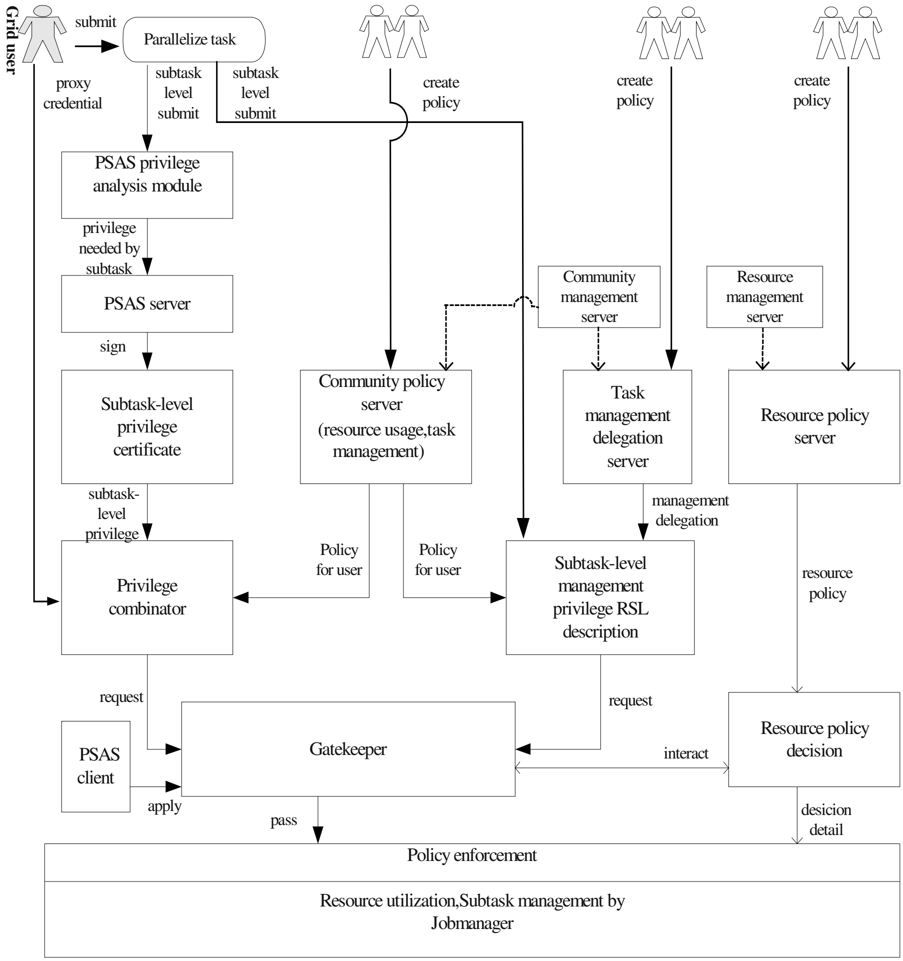


Fig. 1. PSAS architecture overview.

Subtask-level authorization module concentrates on minimizing privileges of tasks by decomposing the parallel task and analyzing the access requirement. To further conform to the least-privilege principle, a few traditional methods restrict the privileges via the delegation of users themselves rather than the architecture, moreover, they only restrict privileges of a whole specific task, not for its constituents (such as subtask). In engineering and scientific computation application, a task is commonly large-scale. It need be divided into many subtasks, and then be distributed to many relevant nodes and run in parallel. Whilst, even though the task is the same, privileges required for distinct subtasks may differ according to operations of these subtasks. By the parallelization and analysis of the task, PSAS can obtain the task's subtasks and relevant required privileges: subtask-level privilege pair. The privileges indicate the information about associated subtask required access to resources at certain nodes. Each task has a subtask-level privilege certificate for recording subtask and privilege pair. An example of this certificate is shown in Figure 2. To prevent malicious third party from tampering with a subtask-level privilege certificate, a trusted third party (PSAS server) signs the certificate. To control a subtask's process and verify the subtask-level privilege certificate, PSAS client will run during resource utilization.

```
Task_main()
    {
  subtask1(){
        (code1)
        }
  subtask2(){
        (code2)
        }
     ...
  subtaskn(){
        (coden)
        }
  subtaskm(){
        (codem)
        }
 parallelize{
       subtask1 ;
       subtask2 ;
        ...
       subtaskn ;
       }
 subtaskm;
    }
```

```
Task_main()
    {
  task right0
   subtask1 right1;
   subtask2 right2;
      ...
   subtaskn rightn;
  subtaskm rightm;
     ...
    }
```

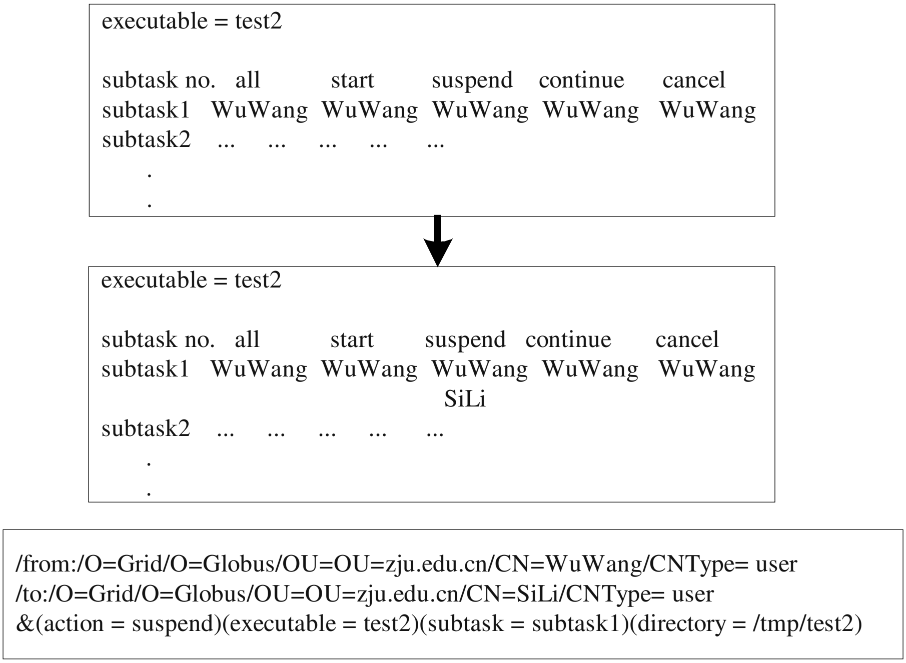
Fig. 2. An example of task and subtask-level privilege certificate.

Subtask-level authorization module contains PSAS privilege analysis module, PSAS server, privilege combinator (shared by community authorization mechanism) and PSAS client.

Community authorization module addresses community authorization mechanism for community member. In Globus, "grid-map" mechanism is conducted, but it is neglected that grid collaboration brings out common rules about privilege for resource usage, resource permission, and so forth, which makes grid security fall into shortage of adequate availability. PSAS architecture imposes similar CAS [8] mechanism with a combination of traditional grid user proxy credential and CAS, as well as the task management policy is added into the community policy server. Two trusted third parties (a community management server and a resource management server) and two policy servers (a community policy server and a resource policy server) are exercised. A community management server is responsible for managing the policies that govern access to a community's resources, and a resource management server is responsible for managing the policies that govern resource permission to grid users.

Two policy servers store the policy for community policy and resource policy respectively. The ultimate privileges of a grid user are formed by the relevant proxy credential and the policy for this user, and the actual rights need accord with resource policy by resource policy decision module during policy enforcement. The policy servers are built or modified by the community administrators or certain specific users.

Community authorization module is composed of a community management server, a resource management server, a community policy server, a resource policy server, privilege combinator, and resource policy decision module.



executable = test2

subtask no.	all	start	suspend	continue	cancel
subtask1	WuWang	WuWang	WuWang	WuWang	WuWang
subtask2	...	...	...	...	...
.					
.					

executable = test2

subtask no.	all	start	suspend	continue	cancel
subtask1	WuWang	WuWang	WuWang SiLi	WuWang	WuWang
subtask2	...	...	...	...	...
.					
.					

```
/from:/O=Grid/O=Globus/OU=OU=zju.edu.cn/CN=WuWang/CNType= user
/to:/O=Grid/O=Globus/OU=OU=zju.edu.cn/CN=SiLi/CNType= user
&(action = suspend)(executable = test2)(subtask = subtask1)(directory = /tmp/test2)
```

Fig. 3. A task management delegation and the relevant change of the subject list.

```
/O=Grid/O=Globus/OU=OU=zju.edu.cn/CN=WuWang/CNType= user:
&(action = start)(executable = test1)(subtask = subtask1)(directory = /tmp/test1)(count<2)
&(action = start,suspend,cancel)(executable = test2)(subtask = subtask1)(directory = /tmp/test2)(count<4)

/O=Grid/O=Globus/OU=OU=zju.edu.cn/CN=Grid administrator/CNType= group:
&(action = all)(executable = all)(subtask = all)(directory = /tmp)
```

Fig. 4. An example of task management description.

Task management authorization module is responsible for managing privilege of task management, authorization to task management and related works. In dynamic grid environments, there are many long-lived tasks, for which static methods of policy management are not effective. Users may also start jobs that shouldn't be under the domain of the VO. Since going through the user who has submitted the original job may not always be an option, the VO wants to give a group of its members the ability to manage any tasks using VO resources. This module imposes a community task management policy and task management delegation to describe rules of task management, and both of the two mechanisms are beneficial to flexible task management in an expressive way. A community task management policy denotes the rules of task management in the whole community, and it is combined into the community policy server. Task management delegation server records a variety of management delegation relations among community users. Once the delegation relation is formed, this task will be able to be managed by the delegate user at its runtime. A community management server is responsible for authenticating task management delegation. To keep compatibility with special task management, the subject list only consists of the owner of the task by default. Figure 3 shows a task management delegation and the change of the subject list for task management privilege to this delegation. Subtask-level privilege management RSL description server produces the task management description, which is expressed by extending the RSL set of attributes. An example of task management description is shown in Figure 4.

Task management authorization module includes a community task management policy, task management delegation server and Subtask-level privilege management RSL description server.

4 Context-Aware Authorization

Based on the PSAS Architecture, the dynamic authorization is able to complement with a low overload; meantime, little impact is enforced on grid computation oriented to scientific and engineering. The main idea is that actual privileges of grid users are able to dynamically adapt to their current context. This work is similar to the study in the literature [13]; however, our context-aware authorization is exercised by constructing a multi-value community policy. The approach is completed according to the following steps:

1. Rank the privileges belonging to each item in a traditional community policy. We divide the privileges of a community member into three sets of privileges: Fat Set, medium Set and thin Set. Fat Set is rich set with the full privileges of this community member, and is suited for the best context at runtime; for example, at some time, the context is fully authorized nodes with least resources utilized, then the node will be able to provide its user most privileges. Medium Set is a set of privileges decreased, and Thin Set is the least privilege set for the community member. When Thin Set is enforced, the subtask belonging to this community member will use the less resources (i.e. less memory, less CPU cycles, etc) or be canceled.
2. Construct a multi-value community policy. After finishing the above, we must rebuild the community policy. To keep compatible with the traditional policy, we

only add two sub-items below each item, and the two sub-items are inserted for medium Set and thin Set, respectively. We apply the policy language defined by the literature [15], and introduce two new tags “$1” and “$2” as the respective beginning statement. An example of a policy of resource usage and task management is shown in Figure 5. The statement in the policy refers to a specific user, Wu Wang, and in the first item, it states that he can “start” and “cancel” jobs using the "test1" executables; The rules also place constraints on the directory “/usr/test1” and on the count “<4”. In its multi-value community policy, corresponding Fat Set maps to the first item without the change of privileges, its Medium Set and Thin Set respectively map to the next two sub-items below the first item, and their authorizations are changed. For instance, in its corresponding Thin Set, the empty statement prevents him from doing any control using the "test1" executables, the statement “memory<20” places a constraint of used memory <20M, and the statement “directory=/tmp/test1” places a constraint of resource usage.

```
/O=Grid/O=Globus/OU=OU=zju.edu.cn/CN=WuWang/CNType= user:
&(action = start,cancel)(executable = test1)(subtask = subtask1)(directory = /usr/test1)(count<4)
&(action = start,suspend,cancel)(executable = test2)(subtask = subtask2)(directory = /usr/test2)(count<6)
```

```
/O=Grid/O=Globus/OU=OU=zju.edu.cn/CN=WuWang/CNType= user:
&(action = start,cancel)(executable = test1)(subtask = subtask1)(directory = /usr/test1)(count<4)
 $1(action = cancel)(executable = test1)(subtask = subtask1)(directory = /usr/test1)(count<2)(memory<100)
 $2(action = )(executable = test1)(subtask = subtask1)(directory = /tmp/test1)(count<2)(memory<20)
&(action = start,suspend,cancel)(executable = test2)(subtask = subtask2)(directory = /usr/test2)(count<6)
 $1(action = start,cancel)(executable = test2)(subtask = subtask2)(directory = /usr/test2)(count<4)
 $2(action = cancel)(executable = test2)(subtask = subtask2)(directory = /tmp/test2)(count<2)(memory<20)
```

Fig. 5. A traditional policy and its corresponding multi-value community policy.

3. The context-aware authorization is conducted via the above multi-value community policy at runtime. As shown in Figure 6, the model uses a Context Agent as an entity to sense the context information. The Transition Controller accepts the trigger from associated Context Agent and makes a decision of transition of privilege set. In addition to these common policies, the Community Policy Server contains transition policy, as a rule of state transition, and event policy, as a rule of sense event. For example, when the memory of hosting node becomes exhausted, the event notifies Context Agent and let it sense and trigger the Transition Controller.

5 Enforcement Mechanisms

Enforcement of fine-grained access rights is defined as the limitation of operations performed on resources or tasks/subtasks by a user to those permitted by an authoritative entity. Based on the Globus Toolkit 2.2, PSAS architecture implements the subtask-level authorization, flexible task management, and context-aware authorization.

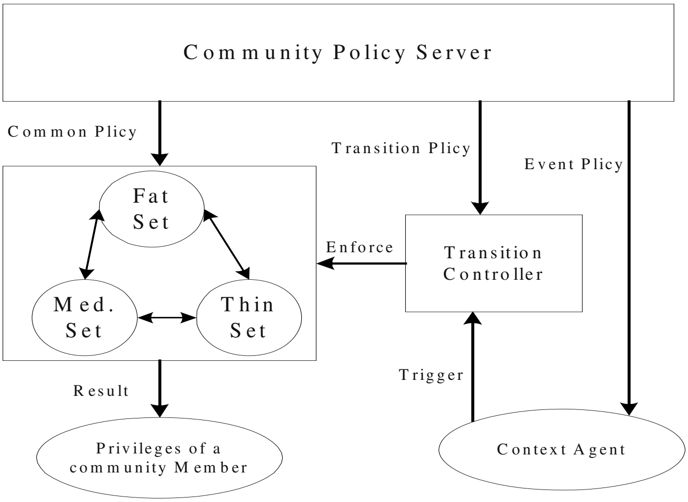


Fig. 6. The context-aware authorization model.

To implement task management authorization module, PSAS creates a job management controller- a component by extending jobmanager in GRAM. When the jobmanager parses users' job descriptions, the job management controller parses and evaluates subtask-level privilege certificate, and makes a decision of certain task management permission. The job management controller integrates with the jobmanager by an authorization callout API. The callout passes the relevant information to the job management controller, such as the credential of the user requesting a remote job, the action to be performed (such as start or cancel a job), a unique job identifier, and the job description expressed in RSL. The job management controller responds by the callout API with either success or an appropriate authorization error. This call is made whenever an action needs to be authorized; that is, it happens before creating a job manager request, and before calls to cancel, query, and signal a running job. PSAS extends the GRAM protocol to return authorization errors with reasons of authorization denial as well as authorization system failures. All of task management delegations and community policies are described in complex task management cases, and they are translated into a RSL regular description by Subtask-level privilege management RSL description server.

To enforce subtask-level authorization module, PSAS employs the proxy certificate with an extension field in the form of standardized X.509 v2 attribute certificates, and makes signed subtask-level privilege certificate embedded into the proxy certificate's extension field. At the same time, PSAS architecture modifies the Globus gatekeeper and jobmanager to put subtask-level authorization into practice. For PSAS client needs to manage a subtask's process and checks whether the subtask is running according to the subtask-level privilege certificate, the GASS library is modified to communicate with PSAS client, and the relevant callout APIs are created. After mutual authentication between a user and a resource, the resource obtains subtask-level privilege certificate located in the proxy certificate's extension field. Then the subtask-level privilege certificate is verified and finds out job identifier. Finally, PSAS client guarantees a subtask's process according to the subtask-level privilege certificate by interacting with GASS.

Community authorization module in PSAS is executed as in CAS [8]. The GSI delegation feature is extended to support rich restriction policies in order to allow grantors to place specific limits on rights that they grant. PSAS employs extensions to X.509 Certificates to carry out restriction policies. However, there exist some differences between CAS and PSAS in the community authorization module. That is, the CAS uses restricted proxy credentials to delegate to each user only those rights granted by the community policy; but the latter regards the ultimate privileges as a privilege combination of the "restricted proxy credentials" and the subtask-level privilege certificate. Proxy credentials are separated from identity credentials. The identity credentials are used for authentication, while the proxy credentials are used for authorization. That makes the PSAS architecture more flexible. Similar to the previous cases, the GASS library is modified to implement a policy evaluation, and the relevant callout functions are designed for call in the Globus gatekeeper. To implement the dynamic context awareness, Context Agent applies a context toolkit described in the literature [18].

6 Conclusions and Future Work

In this paper, we propose a Parallelized Subtask-level Authorization Service (PSAS) architecture to fully secure applications oriented to engineering and scientific computing. This type of task is generally large-scale and long-lived. It needs to be divided into many subtasks run in parallel, and these subtasks may require different privileges. The minimization of the privileges is conducted by decomposing the task and re-allotting the privileges required for each subtask with a subtask-level privilege certificate. With the aid of Context Agent, a multi-value community policy for resource usage and task management enables the context-aware authorization in addition to separating proxy credentials from identity credentials. The delegation mechanism collaboratively performs the authorization delegation for task management together with a relevant management policy. To enforce the architecture, the authors have extended the RSL specification and the proxy certificate and have modified the Globus gatekeeper, jobmanager and the GASS library to allow authorization callouts. The authorization requirement of an application is effectively met in the presented architecture.

At present, the PSAS architecture is only a prototype, and many issues need to be solved. So we plan, firstly, to improve the PSAS architecture in practice via complex applications, and secondly, to further study the policy based context-aware authorization of resource usage and task management based on performance metrics.

Acknowledgements. The authors wish to thank the National Natural Science Foundation of China for the National Science Fund for Distinguished Young Scholars under grant Number 60225009. We would like to thank the Center for Engineering and

Scientific Compu-tation, Zhejiang University, for its computational resources, with which the research project has been carried out.

References

1. I. Foster, C. Kesselman, and S. Tuecke, The Anatomy of the Grid: Enabling Scalable Virtual Organizations, *International Journal of Supercomputer Application*s, 15(3): pp.200-222, 2001.
2. A. Grimshaw, W. A. Wulf, et al., The Legion Vision of a Worldwide Virtual Machine, *Communications of the ACM*, 40(1): 39-45, January 1997.
3. I. Foster and C. Kesselman. Globus: a metacomputing infrastructure toolkit, *International Journal of Supercomputer Applications*, 11(2): 115-128, 1997.
4. S. Tuecke, et al., Internet X.509 Public Key Infrastructure Proxy Certificate Profile. 2002.
5. I. Foster, C. Kesselman, G. Tsudik, S. Tuecke, A Security Architecture for Computational Grids, *Proc. of 5th ACM Conference on Computer and Communications Security Conference*, 1998.
6. L. Kagal, T. Finin, and Y. Peng, A Delegation Based Model For Distributed Trust, *IJCAI-01 Workshop on Autonomy*, Delegation, and Control, 2001.
7. J. R. Salzer and M. D. Schroeder, The Protection of Information in Computer Systems, *Proc. of the IEEE*, 1975
8. L. Pearlman, V. Welch, et al., A Community Authorization Service for Group Collaboration, *Proc. of the 3rd IEEE International Workshop on Policies for Distributed Systems and Networks*, 2002.
9. W. Johnston, S. Mudumbai, et al., Authorization and Attribute Certificates for Widely Distributed Access Control, *Proc. of IEEE 7th International Workshops on Enabling Technologies: Infrastructures for Collaborative Enterprises*, 1998.
10. R. Alfieri, et al., VOMS: an Authorization System for Virtual Organizations, *Proc. of the 1st European Across Grids Conference*, 2003.
11. M. Lorch, D. B. Adams, et al., The PRIMA System for Privilege Management, Authorization and Enforcement in Grid Environments, *Proc. of the 4th International Workshop on Grid Computing*, 2003
12. M. Lorch and D. Kafura, Supporting Secure Ad-hoc User Collaboration in Grid Environments, *Proc. of the 3rd IEEE/ACM International Workshop on Grid Computing*, 2002.
13. G. Zhang and M. Parashar, Dynamic Context-aware Access Control for Grid Applications, *Proc. of the 4th International Workshop on Grid Computing*, 2003.
14. R. Sandhu, E. Coyne, et al., Role-based Access Control Models, *Proc. of the 5th ACM Workshop on Role-Based Access Control*, 2000
15. K. Keahey, V. Welch, et al., Fine-Grain Authorization Policies in the Grid: Design and Implementation, *Proc. of the1st International Workshop on Middleware for Grid Computing*, 2003.
16. S. Kim, J, Kim, S. Hong, et al., Workflow-based Authorization Service in Grid, *Proc. of the 4th International Workshop on Grid Computing*, 2003.
17. M. Romberg, The UNICORE Architecture: Seamless Access to Distributed Resources, *Proc. of the 8th IEEE International Symposium on High Performance Distributed Computing*, 1999.
18. A. K. Dey, G. D. Abowd, The Context Toolkit: Aiding the Development of Context-Aware Applications, *Proc. of Human Factors in Computing Systems: CHI 99*, 1999.

Validating Word-Oriented Processors for Bit and Multi-word Operations

Ruby B. Lee, Xiao Yang, and Zhijie Jerry Shi

Princeton Architecture Laboratory for Multimedia and Security (PALMS)
Princeton University
{rblee, xiaoyang, zshi}@princeton.edu

Abstract. We examine secure computing paradigms to identify any new architectural challenges for future general-purpose processors. Some essential security functions can be provided by different classes of cryptography algorithms. We identify two categories of operations in these algorithms that are not common in previous general-purpose workloads: bit operations within a word and multi-word operations. Both challenge the basic word orientation of processors. We show how very complex bit-level operations, namely arbitrary bit permutations within a word, can be achieved in O(1) cycles, rather than O(*n*) cycles as in existing RISC processors. We describe two solutions: one using only microarchitecture changes, and another with Instruction Set Architecture (ISA) support. We generalize our solutions to define *datarich execution* with MOMR (Multi-word Operands Multi-word Result) functional units. This can address both challenges, leveraging available resources in typical processors with minimal additional cost. Thus we validate the basic word-orientation of processor architectures, since they can also provide superior performance for both bit and multi-word operations needed by cryptographic processing.

1 Introduction

The dependence on the public Internet and wireless networks in modern society poses a growing need for secure communications, computations and storage. To provide basic security functions like data confidentiality, data integrity, and user authentication, different classes of cryptographic algorithms can be used with security protocols at network, system or application levels. Not only network transactions need to be protected, all data and programs may also need these security functions. As secure computing paradigms become more pervasive, it is likely that such cryptographic computations will become a major component of every processor's workload. Understanding the new requirements of secure information processing is critical for the design of all future processors, whether general-purpose, application-specific or embedded. In this paper, we especially target the needs of high performance microprocessors.

Basic security functions include confidentiality, integrity and authentication. Confidentiality of messages transmitted over the public networks, and of data stored in disks, can be achieved by encrypting the data, using symmetric-key cryptography algorithms such as DES [1], and AES [2]. Data integrity, where data is not changed in

P.-C. Yew and J. Xue (Eds.): ACSAC 2004, LNCS 3189, pp. 473–488, 2004.

transit or in storage, can be accomplished with one-way hash functions such as SHA and MD-5 [1]. Authenticating users and devices remotely across the Internet can be accomplished with public-key cryptographic algorithms such as Diffie-Hellman and RSA [1]. They also allow digital signatures and the exchange of a shared secret key across the Internet.

We observe two categories of new requirements imposed by these three classes of cryptographic algorithms: bit-oriented operations and multi-word operations. Both challenge the basic word-orientation of modern processors. Symmetric-key cryptography introduces a new requirement: bit-level permutations. Previously, the bit-oriented operations in general-purpose workloads were SHIFT instructions and logical operations like AND, OR, XOR and NOT. These are supported efficiently by simple single-cycle instructions. Public-key cryptography introduces the other new requirement: multi-word arithmetic. While multiword integer arithmetic has been a requirement in previous high-precision integer computations, its need remained relatively low since the basic word size in general-purpose processors has increased from 16 to 32 to 64 bits. Frequent use of public-key cryptography algorithms may significantly increase the need for multi-word arithmetic. For example, multiplication of two 1024-bit operands in RSA involves two 16-word operands, if each word is 64 bits. If a hardwired multiply instruction operates on two words, many such 64-bit multiply instructions are needed, as well as add operations to accumulate the result. Public-key algorithms based on Elliptic Curve Cryptography (ECC) often perform polynomial operations requiring both bit-oriented and multi-word operations.

A key contribution of this paper is the observation that fast cryptographic processing depends on a processor's ability to support both complex bit-level manipulations as well as multiword operations. These requirements have more impact on performance than defining a new instruction or special-purpose functional unit for accelerating a particular cryptographic primitive. They also challenge the atomic word-orientation of processors, since they emphasize bit operations within a word, and operations requiring operands much larger than a word.

A second contribution is showing how arbitrary bit-level permutations can be accomplished very efficiently in only 1 or 2 cycles.

A third contribution is a generalized architectural solution that allows high-performance processors to support *datarich* operations with flexible, multi-word operands and multi-word result (MOMR) functional units. Our generalized solution supports both high performance bit permutations and multi-word operations. Hence, the basic word-orientation of processors is still a good design choice, since both bit-oriented and multi-word oriented operations can also be supported very efficiently.

In Section 2, we describe past work on permutation instructions, including how our recent past work has reduced the time taken to achieve any *n*-bit permutation down from O(*n*) to O(log(*n*)) instructions and cycles. In Section 3, we propose two new architectural methods for further bringing this down to O(1) cycles. One method is purely micro-architectural, and the other involves new ISA. In Section 4, we describe the changes in the datapath and control path needed to implement our two methods. In Section 5, we generalize these two methods to solve the second challenge of achieving multi-word operations efficiently in word-oriented processors. In Section 6, we discuss performance, and conclude in Section 7.

2 Past Work

Past work in accelerating cryptographic processing included many hardware ASIC (Application Specific Integrated Circuit) implementations of specific ciphers. For programmable solutions, new instructions were proposed to accelerate symmetric-key ciphers in general-purpose processors [3], and in cryptographic coprocessors like Cryptomaniac [4] for ciphers used in secure networking protocols. In contrast, we do not propose any specific new instructions in this paper, but rather new methodologies for bringing more data to the functional units. This *datarich* computation is done with very low overhead, utilizing the datapaths and control already provided for superscalar execution found in most microprocessors, including out-of-order superscalar machines.

The datarich methodology allows us to accelerate both bit permutations used for symmetric-key ciphers, and multi-word operations used in public-key ciphers. It allows us to achieve one of our major contributions: performing arbitrary bit permutations in 1 or 2 cycles – a significant improvement over our recent past work achieving O(log(n)) instructions and cycles [5], which we describe further below.

Performing bit-level permutation has been a hard problem for word-oriented general-purpose processors. Previously, processors only supported a very restricted subset of bit permutations known as rotations. Here, every bit in the n-bit word is moved by the same shift amount, with wrap-around. While some n-bit permutations can be achieved with fewer instructions, allowing arbitrary, data-dependent n-bit permutations is very slow. Conventional logical and shift instructions take O(n) cycles to achieve any one of n! permutations [5]. Alternatively, table lookup methods can be used, but this is limited to a few fixed permutations due to the high memory space requirement, and cache misses cause performance degradation.

More recently, permutation instructions have been introduced into certain microprocessors as multimedia ISA extensions to handle the re-arrangement of subwords packed in registers. Examples are MIX and PERMUTE in HP's MAX-2 [6], VPERM in Motorola's AltiVec [7], and MIX and MUX in IA-64 [8]. However, these instructions can only handle subword sizes down to 8 bits. They do not provide a general solution for performing arbitrary bit-level permutations efficiently.

Very recently, researchers have tackled the general bit permutation problem, and defined new permutation instructions that can achieve any n-bit permutation with only log(n) instructions. Several approaches were proposed. The CROSS [9] and OMFLIP [10] permutation instructions each performs the equivalent function of two stages of a "virtual" interconnection network. A sequence of log(n) CROSS or OMFLIP instructions can build a 2log(n)-stage virtual network that can achieve any one of the n! permutations. Another approach was the GRP instruction [11], which partitions the data bits into two groups. At most log(n) GRP instructions are sufficient to achieve any one of n! permutations [11]. A third approach involves specifying the order of the indices of the source bits in the permuted result. Examples are PPERM[11], and SWPERM and SIEVE [12]. The XBOX instruction [3] is similar to PPERM.

A comparison of CROSS, OMFLIP, GRP, and PPERM is presented in [5]. CROSS, OMFLIP and GRP all achieve arbitrary 64-bit permutations in 6 instructions. PPERM and SWPERM with SIEVE require more than log(n) instructions, but can be executed in as few as 4 cycles on a 4-way superscalar machine. Unfortunately, CROSS, OMFLIP and GRP cannot achieve speedup with superscalar machines, due to the strict data dependency between the sequence of log(n) permutation instructions. Below, we show how this data dependency can be overcome, so that arbitrary 64-bit permutations can be achieved in 1 or 2 cycles, rather than log(n) = 6 cycles.

This paper extends the concepts we presented in [13] with new work on the detailed ISA or microarchitectural changes required, and the detailed implementation in an out-of-order processor.

3 Achieving Arbitrary 64-Bit Permutations in 1 or 2 Cycles

The reason log(n) instructions are needed to achieve any permutation of n bits is because $n$log(n) configuration bits are needed to specify an arbitrary n-bit permutation [5][11]. Since a typical instruction reads up to 2 source operands and produces 1 result, a permutation instruction uses one source operand for the data and the other for n bits of configuration. The intermediate result produced by one permutation instruction is used as the data for the next. Hence, a sequence of log(n) instructions are needed to supply the $n$log(n) configuration bits and the data to be permuted, to achieve any n-bit permutation [5]. If all $n$log(n) configuration bits and the n data bits to be permuted can be specified by a single instruction, then it may be possible to execute any arbitrary n-bit permutation in 1 instruction. Hence, the main performance limiter is the ISA instruction format and the datapaths that support only two n-bit operands per instruction, and a design goal of not having to save states between permutation instructions. This is a reasonable goal since it reduces context-switch and operating system overhead.

Suppose that the latency through the permutation functional unit is not a cycle-time limiter. Then, the problem reduces to the following: how can n(log(n)+1) bits be sent from general registers to a permutation functional unit (PU) in a single instruction? If each register is n bits, this means sending (log(n)+1) register values, or operands, to a functional unit. We propose two methods to solve this problem. Method 1 identifies instruction groups dynamically with microarchitecture techniques; method 2 employs ISA techniques to identify instruction groups statically.

3.1 Datapath, MOMR, and Instruction Groups

We first define some new architectural terms: An *(s,t) functional unit* in a word-oriented processor is a functional unit that takes s word-sized operands and produces t word-sized results. A *standard functional unit* is a (2,1) functional unit.

An *(s,t) datapath* in a word-oriented processor is a datapath where s source buses and t destination buses are connected to functional units. If the datapath contains a register file, it has s read ports and at least t write ports for the results coming from the functional units in one cycle. In general, a k-way multi-issue processor has a (2k,k) datapath, supporting the simultaneous execution of k standard (2,1) functional units each cycle.

A *datarich or MOMR (Multi-word Operands Multi-word Result) functional unit* in a word-oriented processor is a functional unit that requires more than the standard two word-sized operands and one word-sized result.

A sequence of consecutive instructions is called an *instruction group* if the instructions can be executed simultaneously by a datarich (or MOMR) functional unit.

Emerging secure computing paradigms may require extensive execution of algorithms where performance can be improved by the use of datarich functional units. Sometimes datarich functional units improve the performance, other times they improve the cost-performance. Our thesis is that a *k*-way multi-issue processor with a (2*k*,*k*) datapath can accommodate different types of datarich functional units, with relatively minor changes to the pipeline control logic. In the rest of Section 3 and Section 4, we illustrate two methods for datarich MOMR execution, using bit permutation as the example. In Section 5, we generalize datarich MOMR execution to operations with multi-word operands.

3.2 Method 1: Microarchitecture Group Detection

A permutation instruction is defined as follows:

```
PERM rs,rc,rd
```

where rs contains data source, rc contains configuration bits and rd is the result. For example, PERM can be either a CROSS or OMFLIP instruction. Fig. 1(a) shows a 64-bit permutation specified with a sequence of 6 PERM instructions in 2 groups of 3 instructions each. Each group provides the data source and 3 configuration words for a (4,1) permutation unit, PU. The group is dynamically detected, and then its 3 instructions are transformed into 2 "internal" instructions. The first one supplies the data source and one configuration word, and the second one provides the other 2 configuration words. When all the operands are ready, the two “internal” instructions are issued for execution simultaneously on one (4,1) PU. Hence, the 6 instructions can be transformed into 4 internal instructions in 2 groups and executed over 2 cycles. If there are two or more (4,1) PUs, we can pipeline the executions of the 2 groups and achieve a throughput of one permutation per cycle.

3.3 Method 2: New ISA for Group Identification

In the second method, we enhance the conventional RISC instruction encoding with 2 new subop bits, gs and gc, for identifying instructions which start a group (gs=1) or continue a group (gc=1). The meanings of these 2 bits are shown in .

The permutation instruction is defined as:

```
PERM,subop  rs1,rs2,rd
```

where subop contains the gs and gc bits. If gs is set, the instruction is the first in a 2-instruction group, supplying the data word and one configuration word to the (4,1) PU. If gc is set, it is the second instruction in a group, supplying 2 configuration words for the (4,1) PU. We specify this 2-instruction group as follows:

```
PERM,gs   rs,rc1,rd
PERM,gc   rc2,rc3,rd
```

Unlike method 1, method 2 does not need the dynamic group detection and instruction transformation. It also helps reduce static code size, since only 4 permutation instructions (rather than 6) are required in the program. Similar to method 1, when all the source operands for the 2 grouped instructions are ready, they are issued for execution together on one (4,1) PU.

(a) **Original** (b) **Intermediate**

```
PERM rs,rc1,rd          PERM,c    rs, rc1,rd
PERM rd,rc2,rd    →     PERMcont rc2,rc3,rd
PERM rd,rc3,rd

PERM rd,rc4,rd          PERM,c    rd, rc4,rd
PERM rd,rc5,rd    →     PERMcont rc5,rc6,rd
PERM rd,rc6,rd
```

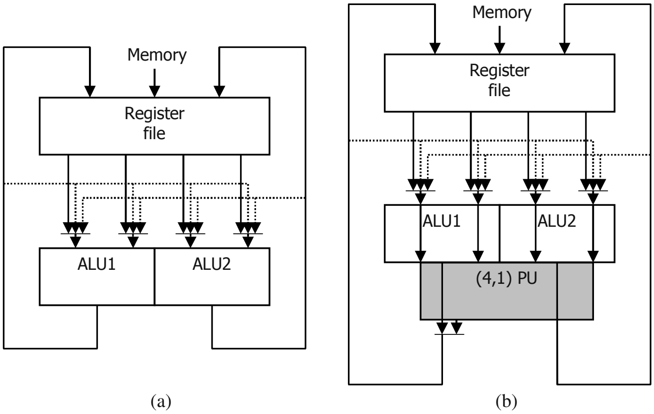
Fig. 1. Instruction transformation for method 1

Table 1. Meanings of gs and gc bits

gs=0, gc=0	Normal instruction, not part of a group
gs=1, gc=0	First instruction in a group
gs=0, gc=1	Continuation instruction in a group
gs=1, gc=1	Reserved

4 Microarchitectural Changes

In this section, we show how either of the two above methods can leverage the resources already present in a superscalar processor, with minimal additional cost. We first describe a typical superscalar processor in Section 4.1, then detail changes that must be made to its datapath and control path in Sections 4.2 and 4.3, respectively.



(a) (b)

Fig. 2. (a) Standard 2-way superscalar processor datapath; (b) with a (4,1) PU added

4.1 Baseline Microarchitecture

Fig. 2(a) shows a standard 2-way superscalar RISC processor with a (4,2) datapath, i.e., 4 register read ports, 2 write ports and associated data buses and bypass paths.

Fig. 3 shows the pipeline frontend of a generic out-of-order superscalar processor [14]. A block of instructions is fetched from the instruction cache. These instructions are then decoded and their operands renamed (to physical registers to eliminate register-name dependencies) before entering the issue window. They will be issued for execution when all their source operands and required functional units become available (wakeup and select stage). Certain stages of the pipeline may take multiple cycles. For an in-order issue processor, there are no rename or select stages.

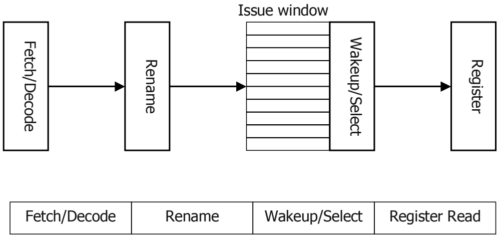


Fig. 3. Generic out-of-order superscalar processor pipeline frontend

4.2 Changes to the Datapath

Fig. 2(b) shows a (4,1) permutation unit (PU) added to a standard (4,2) datapath of a 2-way superscalar processor. Fig. 4 shows an implementation of the PU based on the butterfly network. There are 2 separate PUs, one contains a 6-stage butterfly network and the other contains an inverse butterfly network. In a 2-way processor, we have both of them in the datapath, but only one PU is used at a time, resulting in a 2-cycle latency and a throughput of one permutation per 2 cycles. In a 4-way or wider processor, we can use both of them in parallel. Then we can pipeline the permutation operation and achieve one permutation per cycle throughput (Fig. 5).

Inclusion of a datarich (4,1) MOMR functional unit in a 2-way superscalar processor causes minimal datapath overhead of one additional result multiplexer. All the expensive register ports, data buses and bypasses required have already been provided by the (4,2) datapath of the 2-way superscalar machine. Similarly, for the inclusion of two (4,1) MOMR units in a 4-way superscalar processor. Two (4,1) PUs leveraging the (8,4) datapath of a 4-way superscalar machine are sufficient to achieve the ultimate performance of a different 64-bit permutation every cycle. A key benefit of our solution is leveraging the existing resources of today's microprocessors, essentially all of which are at least 2-way superscalar.

4.3 Changes to the Control Path

We now show that even the required control path changes are minimal. Method 1, which uses the microarchitecture to detect a sequence of dependent instructions that

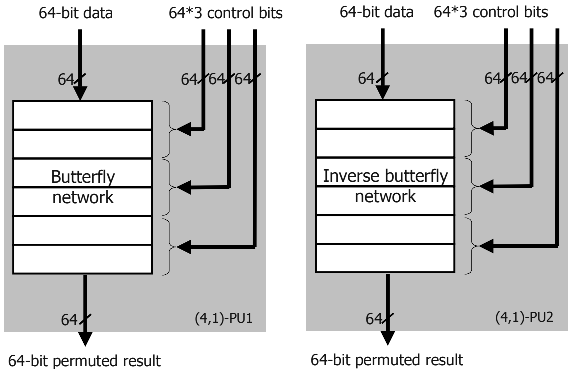


Fig. 4. One implementation of (4,1) permutation FU

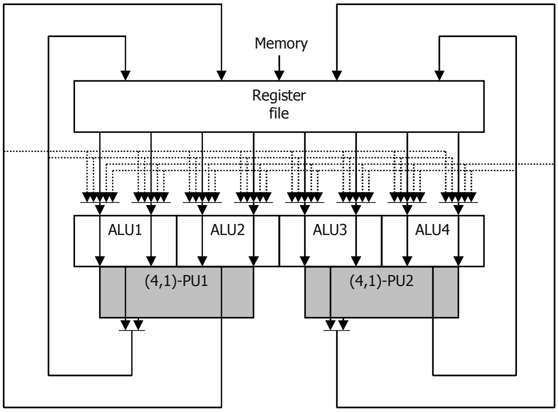


Fig. 5. Two (4,1) permutation FUs in 4-way superscalar processor

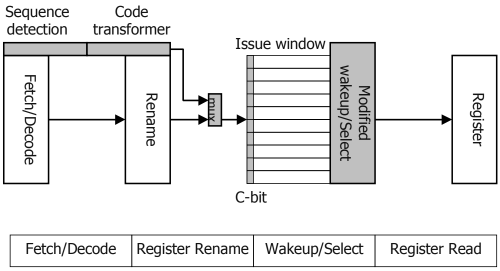


Fig. 6. Modified superscalar processor pipeline frontend

can be executed together, requires some modifications to the pipeline control frontend as shown in Fig. 6. The sequence detection unit detects sequences of 3 permutation instructions. These sequences are then transformed to groups of 2 instructions by the code transformer. The muxes pick the correct inputs to the issue window between the original instructions and the transformed instructions. A 1-bit field reserved for a C-bit is added to each entry in the instruction window to denote whether the corresponding instruction and the following one are in a group. The wakeup/select logic is also modified so that the 2 grouped instructions can be woken up and executed together. For method 2, since instruction groups are explicitly identified by instruction subop bits, the sequence detection unit, code transformer and muxes are not needed. The rest of the control path is the same as for method 1.

Group sequence detection. Dynamic instruction group detection is needed only by method 1. The group sequence detection unit (Fig. 7) recognizes 3 consecutive permutation instructions in a fetch block that satisfy the following two criteria: they have the same opcodes and the data source operand in a permutation instruction is the result of the previous permutation instruction. It then sets C-bits for the first instruction of the detected group sequence. For simplicity, sequences residing in two fetch blocks are not recognized to avoid keeping additional states.

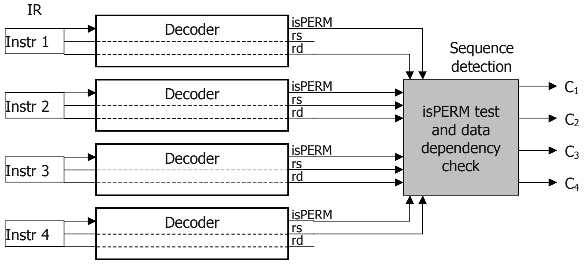


Fig. 7. Functions of sequence detection unit

Instruction transformation. The code transformer is also needed only by method 1. It transforms the group of 3-instruction sequences into 2-instruction sequences (see Fig. 1). The 2 new instructions are generated according to the C-bits produced by the sequence detection unit and the renamed operands of the original 3 instructions. The code transformer replaces the data operand in the second instruction with the configuration operand from the third instruction before discarding the third instruction. Then, it updates the C-bits in the newly generated instructions. An instruction that has its C-bit set starts a group. Grouped instructions are adjacent in the issue window. Fig. 8 shows the functions of the code transformer.

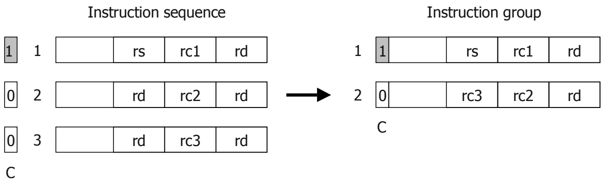


Fig. 8. Code transformer transforms 3-instruction sequence to 2-instruction group

Instruction wakeup. Fig. 9 shows the modified wakeup logic needed by both methods to wake up the 2 grouped instructions together. This is necessary because otherwise the 2 instructions might be issued separately, producing the wrong result.

Previously, an instruction is ready to issue when both of its source operands are ready. The modified wakeup logic ensures that grouped instructions become ready only when all the source operands in the group are ready.

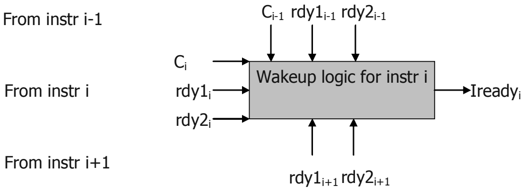
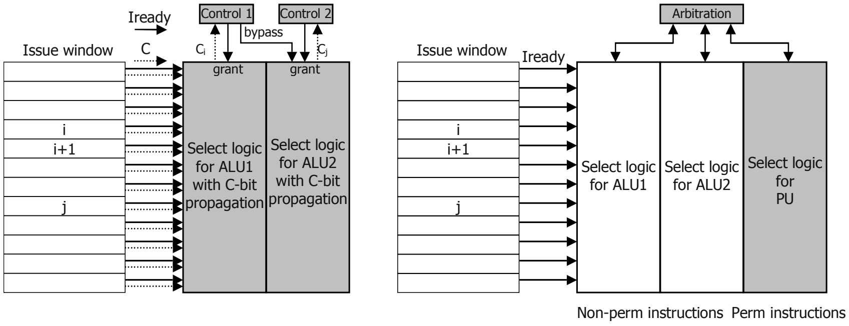


Fig. 9. Modified instruction wakeup logic

Instruction select. We can modify the select logic for ALU1 and ALU2 to handle the permutation unit as well. This is achieved by adding C-bit propagation to the original select logic for ALU1 and ALU2 and 2 small control units, as shown in Fig. 10(a). Assume the select logic for ALU1 selects instruction i. The control unit 1 tests the C-bit of i. If i's C-bit is set, then grant both instruction i and i+1 and bypass the select logic for ALU2. Otherwise grant i and proceed to select logic for ALU2. Suppose instruction j is selected for ALU2. The control unit 2 then tests the C-bit of j. We grant instruction j only if j's C-bit is not set.



(a) (b)

Fig. 10. (a) Select logic with modifications on the original select logic for ALU1 and ALU2; (b) Select logic with new set of logic for PU

Alternatively, we can add a new set of select logic for the PU, which deals only with instructions with C-bits set, while the select logic for ALU1 and ALU2 deals with normal instructions. The arbitration unit picks the result of either the select logic for ALU1 and ALU2 or the new select logic. (see Fig. 10(b)).

If there are multiple issue queues, such as proposed in [14], we can devise an instruction steering method so that the 2 permutation instructions in a group are dispatched to the same queue. This is easy to achieve because the 2 grouped instructions are adjacent. If the C-bit of the instruction at the head of a queue is set, we grant this instruction together with the following one.

4.4 Complexity and Delay of Control Path Modifications

The modifications to the control path consist of a small amount of combinatorial logic, estimated at a few thousand gates for a 4-way superscalar processor. As comparison, the issue logic of the Compaq Alpha 21264 processor, a 4-way superscalar RISC processor, contains about 141000 transistors [15], making the complexity of our modifications negligible.

In terms of delay, the sequence detection unit and the code transformer run in parallel with the decode and rename logic. Due to their simple functions, they should have no impact on the processor cycle time. Since the wakeup and select logic are already in the critical path for back-to-back executions of dependent instructions, our modifications may increase the cycle time. However, many methods have been proposed to reduce the latency of issue logic by either simplifying the instruction issue logic [14][16][17][18], or breaking wakeup/select to multiple stages [19][20] in order to achieve fast instruction scheduling. By incorporating these methods, we can integrate our modifications without affecting the processor cycle time.

5 Generalization to Multi-word Operations

We define *multi-word operations* as operations that use more than 2 word-sized operands and produce more than 1 word-sized result, i.e., they are operations that could use datarich MOMR functional units. Arbitrary bit permutation is one example of multi-word operations since the configuration bits span multiple words. Other multi-word operations include the multiplication of two 16-word operands in a 64-bit processor, for a public key algorithm like RSA using 1024-bit keys. If larger hardware multipliers can be accommodated within a high performance microprocessor, we can speed up the multiword multiplication by producing longer (and fewer) partial products with each instruction, resulting in fewer instructions needed to accumulate the partial products to get the final result. In particular, if the implementation can afford larger multipliers, we want to eliminate the ISA restriction of only performing the multiplication of two word-sized operands per instruction.

5.1 Multiplication of Multi-word Operands

The use of MOMR methods for speeding up *n*-bit permutations was described in Sections 3 and 4. We now describe how MOMR methods may be used to accelerate the multiplication of multi-word operands. Let the original multiply instructions be:

```
MUL,L ra,rb,rc
MUL,H ra,rb,rd
```

Two 64-bit registers ra and rb are multiplied together to generate the low and high 64 bits of the result in rc and rd, in successive instructions. Actually, both halves of the product are generated by the same hardware multiplier at the same time, and it is only because of the ISA restriction of one word-sized result per instruction that two separate instructions have to be used to generate the double-word result. If a (2,2) instruction were available, then these two instructions can be executed together on one multiplier simultaneously. Method 1 can recognize this case at run-time. Method 2 can specify this at compile time with the gs and gc bits:

```
MUL,L,gs ra,rb,rc
MUL,H,gc ra,rb,rd
```

A 2-way supercsalar processor with two (2,1) multipliers can achieve the same performance as a single (2,2) MOMR multiplier, but with twice the area for two multipliers. Hence, MOMR execution is more cost-effective.

Alternatively, an even higher performance microprocessor may be able to afford a 128-bit multiplier. Implemented as a (4,2) MOMR multiplier, we can execute 128-bit versions of the Multiply Low and High instructions, in method 2 as follows:

```
MUL,L,gs ra1,rb1,rc1
MUL,L,gc ra2,rb2,rc2
MUL,H,gs ra1,rb1,rd1
MUL,H,gc ra2,rb2,rd2
```

The first two MUL,L instructions would be executed together as a group on a 128-bit multiplier to generate the low 128 bits of the result. The next two MUL,H instructions generate the high 128 bits of the result. To get the equivalent 256-bit product using only 64-bit multipliers and conventional (2,1) instructions, we need to do 8 multiply's and 5 add's. A 128-bit multiplier can also be used for (4,4) MOMR execution, where all 4 instructions above belong to the same group and are executed together. Larger multipliers can also be used, for even further speedup.

5.2 Datarich MOMR Execution

We now define generalized MOMR or datarich execution. In Table 2 and Table 3, method 1 achieves a microarchitecture solution for MOMR execution, while method 2 yields an ISA solution, where the MOMR operations are explicitly specified with new gs and gc bits in the instruction encoding. The steps in these 2 methods are listed in Table 2. Method 1 requires a more complex group detection unit to recognize all the supported multi-word operations. For method 2, a similar unit is also necessary, but to check the correctness of groups. Examples of the criteria for recognizing or checking instruction groups are given in Table 3. The first 3 columns specify the instructions in the instruction stream, and the last 2 specify the data dependencies they must satisfy.

In order to simplify the architectural solution, we require that instructions to be executed together as a group be consecutive in sequential program order. That is, we are not trying to look through the whole program to find instructions that may be far apart which can be executed together in the same cycle. Rather, we target programs which can be re-compiled, or new programs, so that instructions that can be "grouped" for simultaneous execution are next to each other.

The ISA cost of method 2 is that we must define the gs bit for every instruction that can serve as the start of a multi-word operation and the gc bit for all instructions that can act as continuation instructions in a group. Encoding space may be tight in existing ISAs and one or two unused bits per instruction may not be available.

When there are different instruction groups, the microarchitecture needed to support method 2 is not significantly simpler than in method 1. However, method 2 can specify MOMR execution opportunities that are too difficult for method 1 to recognize dynamically. For example, it takes a long sequence of 64-bit multiply and add instructions to get the result equivalent to the multiplication of two 128-bit operands. With the gs and gc bits, method 2 only needs 4 instructions to specify this operation (last 2 rows in Table 3). Therefore, method 2 can support a broader scope of multi-word operations.

Table 2. Architectual methods for MOMR execution

Steps	Method 1: microarchitecture detected groups	Method 2: ISA specified groups
Group detection: Recognize a small set of pre-defined groups of instructions that can be executed together on a MOMR functional unit in the same cycle.	Need to consult a table like Table 3 to check multiple combinations of opcodes and data dependencies so as to recognize all the supported multi-word operations.	Although groups are already defined in the ISA, still need to consult a table similar to that for method 1 to determine whether gs and gc define legitimate (and complete) groups.
Instruction transformation: Transform the instructions in a group to fewer instructions with some operand register re-packaging, if necessary.	For different multi-word operations, different transformations are needed.	Set C-bits in microarchitecture according to gs and gc bits in instructions.
Wakeup and select: Wake up and select the instructions to be executed on one MOMR functional unit together	Similar to permutation-only case. The logic shown in Section 4 deals with simultaneous executions of up to 2 instructions. It can be extended if 3 or more instructions are to be issued together. This may result in increased latency and more stages for wakeup and select.	

Table 3. Examples of Group Definitions

	Instr i	i+1	i+2	Dependency criteria	Remarks
Method 1	PERM	PERM	PERM	$rd_i=rs1_{i+1}$ & $rd_{i+1}=rs1_{i+2}$ & $rd_i!=rs2_{i+1}$ & $rd_i!=rs2_{i+2}$ & $rd_{i+1}!=rs2_{i+2}$	Serial dependency No RAW hazard*
	MUL,L PMIN	MUL,H PMAX		$rs1_i=rs1_{i+1}$ & $rs2_i=rs2_{i+1}$ & $rd_i!=rd_{i+1}$ & $rd_i!=rs1_{i+1}$ & $rd_i!=rs2_{i+1}$	Same sources Diff dest for H, L No RAW hazard
Method 2	PERM,gs	PERM,gc		$rd_i!=rs1_{i+1}$ & $rd_i!=rs2_{i+1}$	No RAW hazard
	MUL,L,gs PMIN,gs	MUL,H,gc PMAX,gc		$rs1_i=rs1_{i+1}$ & $rs2_i=rs2_{i+1}$ & $rd_i!=rd_{i+1}$ & $rd_i!=rs1_{i+1}$ & $rd_i!=rs2_{i+1}$	Same source Diff dest for H, L No RAW hazard

* Since a PERM group is composed of 3 serially dependent instructions, there must be data dependencies (RAW hazards) between adjacent PERM instructions. No RAW hazard here means no additional data dependencies other than those required for a serial chain. For multi-word operations (all the rest), no RAW hazard means no RAW data dependencies.

6 Performance

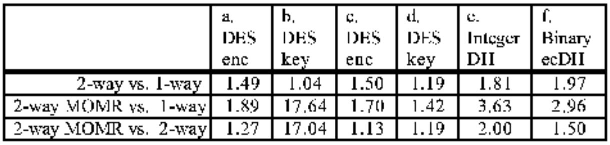
We test two distinct aspects of our new architecture: support for fast bit permutations and for multi-word operations. Table 4 illustrates the performance of our architecture.

For bit permutation, we test DES encryption (DES enc) and round key generation (DES key) with the fastest software program on existing processors which uses table lookup to perform bit permutations (columns a and b). We then test DES using an enhanced ISA that has an OMFLIP permutation instruction [10] added to it (columns c and d). For multi-word operations, we test integer Diffie-Hellman (column e).

We implement these programs using a generic 64-bit RISC processor. First, we obtain the execution time, in cycles, of the programs running on a single-issue processor with one set of (2,1) functional units, including a 64-bit ALU, a 64-bit shifter, a 64-bit permutation unit (for columns c and d), and a 64-bit integer multiplier (for column e). Second, the same programs are executed on a standard 2-way superscalar processor with two sets of (2,1) functional units. The speedup is shown in the first row of Table 4.

Then, we simulate the programs on an enhanced 2-way superscalar processor with one MOMR functional unit. For DES, the MOMR unit is a (4,1) Butterfly permutation unit as detailed in Sections 3 and 4. For DH (columns e), the MOMR unit is a (4,2) multiplier as described in Section 5. This is a 128-bit multiplier, which we are now able to utilize, but could not previously because of ISA limitations in a standard 64-bit superscalar processor. We assume a latency of 3 cycles for a 64-bit multiplier, and 5 cycles for the 128-bit multiplier. Either method 1 or 2 can be used in the DES programs (columns a-d). Method 2 is used for the DH program which is re-coded using the new ISA features to specify grouped instructions with the gs and gc bits. The cache parameters used in the DES simulations are 16 kilobytes L1 data cache and 256 kilobytes L2 unified cache with 10-cycle and 50-cycle miss penalties, respectively.

Table 4. Speedup of execution time



	a. DES enc	b. DES key	c. DES enc	d. DES key	e. Integer DH	f. Binary ecDH
2-way vs. 1-way	1.49	1.04	1.50	1.19	1.81	1.97
2-way MOMR vs. 1-way	1.89	17.64	1.70	1.42	3.63	2.96
2-way MOMR vs. 2-way	1.27	17.04	1.13	1.19	2.00	1.50

The second row of Table 4 shows the speedup of our enhanced 2-way processor with a MOMR functional unit over a single-issue machine. In all cases, our new architecture achieves greater speedup over single-issue execution than the standard 2-way superscalar processor (first row). The third row illustrates the additional speedup provided by our 2-way MOMR architecture over standard 2-way superscalar processors. For DES, the performance gain is very pronounced for key generation (17X speedup in column b), where permutation operations are more frequent than for encryption. The MOMR speedup is less when compared to the enhanced ISAs (columns c and d) than when compared to existing ISAs (columns a and b). This is because the introduction of new permutation instructions (CROSS or OMFLIP) in columns c and d already yields huge speedup over the table lookup method (in columns a and b). The number of instructions for a 64-bit permutation is reduced from over 20 to at most 6, and most of the memory accesses are also eliminated, resulting in much fewer cache misses. Even then, our MOMR execution achieves an additional speedup of 13% to 19% by further reducing the cycles needed for a 64-bit permutation from 6 to 2 cycles.

For the integer DH, there is significant additional speedup of 2X over the standard 2-way superscalar processors. This is because our MOMR architecture allows the inclusion of wider functional units such as 128-bit multipliers. This reduces the overall number of instructions and cycles needed to complete a 1024 by 1024-bit multiplication, which is a primitive operation in the exponentiation function needed by public-key cryptography algorithms.

7 Conclusions

This paper makes several new contributions. First, we identify two categories of bit and multi-word operations as new challenges for word-oriented processor architecture for high-performance cryptographic processing. This insight is more useful from a broad architectural perspective than just picking out special-purpose operations to accelerate.

Second, we present two architectural solutions for achieving arbitrary 64-bit permutations in O(1) cycles. This is a significant result since previously arbitrary *n*-bit bit permutations took O(*n*) cycles. Even with our recent proposals of permutation instructions [5][9][10][11][12], this took at least O(log(*n*)) cycles. We show how a different 64-bit dynamically–specified permutation can be achieved *every cycle* by a 4-way superscalar processor with datarich MOMR execution. Our software solution for achieving permutations is much more powerful than a hardware solution – the latter can only achieve a few statically-defined permutations, while our solution can achieve all possible dynamically-defined permutations. Furthermore, the incremental cost is minimal, since we leverage common microarchitecture trends like superscalar processors. Our result is also significant because it implies that word-oriented processors have no problem supplying very high performance (1 or 2 cycles) for even extremely challenging bit-oriented processing like arbitrary bit permutations. Cryptographers can use bit permutations freely in their new algorithms if microprocessor architectures include these bit permutation instructions.

Third, we define the concepts of datarich MOMR (Multi Operands Multi Result) execution and instruction groups. MOMR functional units can achieve extremely high performance for bit-level permutations as well as multi-word operations, with a single coherent architectural solution. The MOMR feature enables a very flexible extension of standard ISAs to support datarich operations of many flavors. We do not have to decide whether instruction formats of future processors should support (3,1), (4,1), (2,2), (3,2), or (4,2) functional units, all of which are useful for different operations. They can all be supported on a 2-way superscalar machine with minimal changes. Our proposal to base MOMR implementations on the (2*k*,*k*) datapath of a *k*-way superscalar processor gives us the flexibility of supporting all MOMR functional unit sizes covered by these existing datapath resources. We have also shown the control path modifications needed to support MOMR; these are minimal when compared to the complex pipeline control in typical superscalar, out-of-order machines.

Finally, a fourth contribution is the validation of the word as the atomic unit upon which a processor is optimized, since we show how both bit and multi-word operations can be achieved with MOMR execution for either superior performance or enhanced cost-performance.

References

[1] B. Schneier, *Applied Cryptography*, 2nd Ed., John Wiley & Sons, Inc.,1996.

[2] NIST (National Institute of Standards and Technology), "Advanced Encryption Standard (AES) - FIPS Pub. 197", November 2001.

[3] J. Burke, J. McDonald and T. Austin, "Architectural support for fast symmetric-key cryptography", *Proceedings of ASPLOS 2000*, pp. 178-189. November 2000.

[4] L. Wu, C. Weaver, and T. Austin, "CryptoManiac: a fast flexible architecture for secure communication", *Proceedings of the 28th International Symposium on Computer Architecture*, pp. 110-119, June 2001.

[5] R. B. Lee, Z. Shi, and X. Yang, "Efficient permutation instructions for fast software cryptography", *IEEE Micro,* vol. 21, no. 6, pp. 56-69, December 2001.

[6] R. B. Lee, "Subword parallelism with MAX-2", *IEEE Micro*, Vol. 16, No. 4, pp. 51-59, August 1996.

[7] K. Diefendorff et al, "AltiVec extension to PowerPC accelerates media processing", *IEEE Micro*, Vol. 20, No. 2, pp. 85-95, March/April 2000.

[8] "IA-64 application developer's architecture guide", Intel Corp., May 1999.

[9] X. Yang, M. Vachharajani, and R. B. Lee, "Fast subword permutation instructions based on butterfly networks", *Proceedings of SPIE 2000*, pp. 80-86, January 2000.

[10] X. Yang and R. B. Lee, "Fast subword permutation instructions using omega and flip network stages", *Proceedings of the International Conference on Computer Design*, pp. 15-22, September 2000.

[11] Z. Shi and R. B. Lee, "Bit permutation instructions for accelerating software cryptography", *Proceedings of the IEEE International Conference on Application-Specific Systems, Architectures and Processors*, pp. 138-148, July 2000.

[12] J. P. McGregor and R. B. Lee, "Architectural enhancements for fast subword permutations with repetitions in cryptographic applications", *Proceedings of the International Conference on Computer Design*, pp. 453-461, September 2001.

[13] R. B. Lee, Z. Shi, and X. Yang, "How a processor can permute n bits in O(1) cycles", *Proceedings of Hot Chips 14 - A Symposium on High Performance Chips*, August 2002.

[14] S. Palacharla, N. P. Jouppi, and J. E. Smith, "Complexity-effective superscalar processors", *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pp. 206-218, 1997.

[15] J. A. Farell and T. C. Fischer, "Issue logic for a 600-mhz out-of-order execution microprocessor", *IEEE Journal of Solid-State Circuits*, Vol. 33, Issue 5, pp. 707-712, May 1998.

[16] S. Onder and R. Gupta, "Superscalar execution with direct data forwarding", *Proceedings of the 1998 ACM/IEEE Conference on Parallel Architectures and Compilation Techniques*, pp. 130--135, 1998.

[17] D. S. Henry, B. C. Kuszmaul, G. H. Loh, and R. Sami, "Circuits for wide-window superscalar processors", *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pp. 236-247, 2000.

[18] R. Canal, A. Gonzalez, "A Low-complexity issue logic", *Proceedings of the 14th international conference on Supercomputing*, pp. 327-335, 2000

[19] J. Stark, M. D. Brown, and Y. N. Patt, "On pipelining dynamic instruction scheduling logic", *Proceedings of the 33th Annual ACM/IEEE International Symposium on Microarchitecture*, pp. 57-66, 2000.

[20] M. D. Brown, J. Stark, and Y. N. Patt, "Select-free instruction scheduling logic", *Proceedings of the 34th ACM/IEEE International Symposium on Microarchitecture*, pp. 204-213, December 2001.

Dynamic Fetch Engine for Simultaneous Multithreaded Processors

Tzung-Rei Yang and Jong-Jiann Shieh

Department of Computer Science and Engineering, Tatung University

Taipei, Taiwan

Abstract. While the fetch unit has been identified as one of the major bottlenecks of Simultaneous Multithreading architecture, several fetch schemes were proposed by prior works to enhance the fetching efficiency. Among these schemes, ICOUNT, proposed by Tullsen et al. were considered to be a great scheme. The ICOUNT scheme works mainly because it favors the thread which fast moving through the pipeline, thus use the resource effectively. We think it is better letting the thread which tends to have more long latency instructions to get the priority at adequate time since long latency instructions are very likely on program's critical path. We proposed a dynamic fetch scheme which gives the long latency bound thread higher priority while the RUU or LSQ is under low usage. Our motivation is to gain further performance by not only use the resource effectively but also by the urgency of the instructions.

1 Introduction

Simultaneous Multithreading [2, 4, 6, 10, 12] is a processor design that attempts to combine both the hardware features of superscalar and multithreaded processors, and allows instruction-level and thread-level parallelism to be used interchangeably.

Simultaneous multithreaded processor can be roughly divided into two parts. The fetch engine at the front end of the pipeline is composed of the fetch unit, the instruction cache, the decode unit, the register renaming unit and the branch predictor. This part of the processor is responsible for fetching more instructions from multiple threads as much as it can and feeding them to the execution engine which comprises the instruction issue logic, the functional units and the memory hierarchy at the later pipe-stages.

Take a look at the fetch engine, the fetch unit acts a more important role as it used to be. The instructions fetched for delivering to execution engine are now from multiple threads, thus there are likely more independent instructions capable of being issued each cycle. While issuing becomes more efficient, it means that the fetch unit now has to make more effort on fetching more instructions quickly to utilize the shared processor resources. Besides, since the instructions are from different threads

P.-C. Yew and J. Xue (Eds.): ACSAC 2004, LNCS 3189, pp. 489–502, 2004.

now, the fetch unit needs to be smart enough to know which thread to fetch from. In fact, the fetch unit becomes one of the major bottlenecks of the simultaneous multithreading architecture.

Prior works have shown that using intelligent fetch heuristics would do great helps on increasing performance of simultaneous multithreading. It is meant to define a scheme of designating priorities on threads to be fetched, letting fetch unit know which thread the next to fetch from. Through the proposed schemes, ICOUNT [10] proposed by Tullsen et al. has been identified as one of the best schemes of not only the improvement it gained but also the efficiency of implementation.

ICOUNT scheme gives the highest priority to thread which has the fewest instructions in the decode stage, the rename stage, and the instruction queues. It works mainly because the policy favors the threads fast moving through the pipeline. However, we think that favoring the fast moving through pipeline threads might degrade the utilization of shared resources. Further, take the ready queue implementation heuristics into consideration, long latency instructions, such as load instructions and floating point instructions, are very likely on program's critical path. If they do not get their chance to be fetched into pipeline early, they may prevent more other instructions from the thread to be executed.

In order to further utilize the processor resources and gain more improvements, we proposed a fetch scheme based on ICOUNT which aggressively attacks the resource usage by letting more long latency instructions being fetched while the register update unit (RUU) or load/store queue (LSQ) is under low usage. Otherwise, we let the thread with fewest instructions in the decode stage, the rename stage, and the instruction queues has the highest priority as ICOUNT does.

This paper is organized as follows: Section 2 reviews the related works. Section 3 introduces more detailed simultaneous multithreading architecture. Section 4 discusses the fetch scheme of simultaneous multithreaded processor. Section 5 shows the simulation methodology and results. Section 6 presents the conclusion.

2 Related Works

Hirata [4] presented the architecture which considered as predecessors of simultaneous multithreading with thread slots containing an instruction queue, decode unit and program counter. Branch instructions were executed inside the decoding unit and data dependencies were handled using scoreboarding that dispatched ready instructions to standby stations in this architecture. A large register file organized in a bank per thread is used to keep thread contexts.

Tullsen et al. [2, 6, 10, 12] proposed the simultaneous multithreading architecture and firstly implemented it on MIPS R1000 and DEC Alpha platform. They also studied fetch policies for SMT processors and investigated several fetch policies, such as ICOUNT, BRCOUNT, IQPOSN, MISSCOUNT which attempt to improve on the simple round-robin priority policy by using feedback from the processor pipeline. In particular, the ICOUNT fetch policy has been chosen by many researches as their base fetch policy. They further identified the impact of long-latency loads in a simultaneous multithreading processor in [11], and find that it is better to free the resources associated with a stalled thread rather than keep the thread ready to immediately begin execution upon return of the loaded data.

Luo et al. [7] proposed a fetch scheme that uses both fetch prioritizing and fetch gating for simultaneous multithreading processors. Fetch prioritizing sets up fetch priority for each thread based on the number of unresolved low-confidence branches from the thread to find threads that are most likely in their correct paths, while fetch gating prevents fetching from a thread once it has a stipulated number of outstanding low-confidence branches.

Knijnenburg et al. [5] proposed a fetch policy based on a dynamic branch classification mechanism that avoids issuing instructions to the pipeline if the instruction may not belong to the correct execution path. In this way, the resources, such as instruction queues, may be freed from useless wrong path instructions.

El-Moursy et al. [3] proposed a front-end policy that reduces the required integer and floating point issue queue size in simultaneous multithreaded processors. The structure focus on both speed-enhancing and power-saving of the issue queue, that is, they try to reduce the occupancy of the instruction issue queue by trying to limit the unready instructions and data missing instructions in the queue for the same level of performance.

Madon et al. [8] proposed an implementation of the simultaneous multithreaded processor called SSMT (SimpleScacar Multithreaded) using SimpleScalar tool set. The simulator is further enhanced by the Vortex Project [1] at the University of Maryland.

Marr et al. [9] proposed a technique called Hyper-Threading which implements the simultaneous multithreading architecture on modern x86 processors.

3 Simultaneous Multithreaded Processor Architecture

Before breaking into further discussions, we briefly describe the simultaneous multithreading architecture in this chapter.

3.1 Simultaneous Multithreading Architecture

The simultaneous multithreading architecture proposed by Tullsen et al. [6] can be roughly divided into two major parts. The fetch engine at the front end of the pipeline including the fetch unit, the instruction cache, the decode unit, the register renaming unit and the branch predictor is responsible for filling the later pipeline stages with instructions. On each cycle, the fetch unit fetches instructions from multiple threads and fills the instruction cache with them. After decoding, the register renaming logic removes false register dependences by mapping the architectural registers to physical renaming registers.

Instructions are then fed to the execution engine which consists of the instruction issue logic, the functional units, the memory hierarchy, the result forward mechanism, and the reorder buffer. This part of the processor executes the instructions as quickly as their inputs are ready. Processor resources are shared by multiple threads dynamically. Instructions stay in either the integer or floating point instruction queues until their operands become available, and are then issued from these queues to the corresponding functional units.

Conventional superscalar architecture suffers from low instruction level parallelism due to only fetch from one thread at a cycle. Simultaneous multithreading fetches from several multiple threads and shares all major resources to the active threads. In this architecture, instructions from all threads competing for the shared resources each cycle.

3.2 Simplescalar Multithreaded Architecture

While Tullsen et al. construct simultaneous multithreading architecture we discussed last section base on DEC Alpha platform, Madon et al. implement the architecture called SSMT (SimpleScalar Multithreaded) [8] using SimpleScalar tool set which simulates a superscalar architecture on a x86 machine.

The SSMT has been chosen as our base architecture since the hardware requirement of the simulator is quite easy to obtain. The important characteristics of the previous discussed architecture were maintained, such as fetching from multiple threads and issue multiple threads per cycle and dynamic resource sharing, etc. However, there are some differences of the two implementations. For example: the instruction queues of SSMT were divided in different manner: load and store instruction are fed into an independent Load/Store Queue (LSQ), while other instructions are delivered to normal instruction queue. The pipeline stages are different, too.

We will have the simulator discussed more detail in chapter 5.

3.3 Bottlenecks of Simultaneous Multithreading Architecture

Although the simultaneous multithreading architecture dynamically sharing the processor resources to exploit both the thread-level parallelism came from multiple threads and instruction-level parallelism from single thread and better utilizing the resources, there are several bottlenecks identified.

Simultaneous multithreading improves performance in the benefits of dynamic sharing of resources, but it does appear to have some potential drawbacks due to inter-thread contention. Instructions competed for resources now coming from multiple threads instead of a single one puts greater stress to the shared structures such as caches, translation look-aside buffers and branch target buffers than traditional processors do. For example, sharing the cache with multiple threads, that is, partitioning the cache into pieces for threads will eventually reducing the cache space used by each thread, hence decrease the degree of locality and cause cache misses to arise.

Instruction fetching unit is one of the major performance bottlenecks which also widely studied. On one hand, the simultaneous multithreading fetch unit benefits from inter-thread competition for instruction bandwidth by partitioning the bandwidth among threads and finding more useful instructions to fill the issue slot, which is often difficult to fill if there is only one thread to be accessed at a time. On the other hand, dynamic scheduler of simultaneous multithreaded processors which issuing more instructions (from multiple threads) than traditional processors (from a single thread) does put more stress on fetch unit. It must now fetch more quickly to keep pace with the speed that consumed by later pipe-stages.

In order to improve fetch efficiency, the fetch unit must smart enough to determine which thread to fetch from since there may be several threads running at any given time. Several fetch schemes have been proposed to improve the simultaneous multithreading performance.

Another problem is the impact of the long-latency instructions. This happens when the memory-bond threads or threads with high concentration of long-latency instructions fills the instruction scheduling window with instructions that cannot be issued quickly hence prevent other threads to be fetched and even worse, stall the processor. This problem can be solved by either increasing the size of instruction queue or good fetch scheme design.

We are interested in thread priorities while fetching and proposed our fetch scheme which will discuss further in the later chapter.

4 Dynamic Fetch Scheme

Since instructions from different threads are fed into the same shared instruction queues in a simultaneous multithreaded processor, how the instruction fetch unit fills these instruction queues affect the performance of a simultaneous multithreaded processor seriously.

The fetch unit of a simultaneous multithreaded processor generally works as follows: fetch unit selects instructions from multiple threads each cycle. It will try to take instruction from the first thread to fill the fetch bandwidth until it encounters a branch instruction or the end of a cache line. As long as there is still available fetch bandwidth, it will then take instructions from next thread, if any, yielding more issued instructions per cycle and better utilization of the processor's resources.

A priority based fetch scheme is to determine the priorities among threads to let the fetch unit know which the next thread to fetch from.

Previous studies show that the fetch unit becomes one of the major bottlenecks of simultaneous multithreading architecture. We believe a good fetch policy will eventually improves the performance of simultaneous multithreaded processor, and better utilize the resource.

ICOUNT, with which highest priority is given to those threads that have the fewest instructions in the decode stage, the rename stage, and the instruction queues, has been widely used in researches not only because its efficiency but also the simplicity on implementation. It is fast because it gives fast moving through pipeline threads the highest priority.

To our knowledge, there exists some point to argue with. First, long latency instructions, such as load instructions and floating point instructions, are very likely on program's critical path. Actually, ready instruction queues are often designed to give these instructions higher priorities to improve the overall processor performance. Second, an instruction will never get it chance to execute if it is not even being fetched. That is, if a long latency instruction loses its opportunity to be fetched into the pipeline, it will potentially block further instructions. This might decrease the overall performance of execution of multiple threads.

But, if we just give the highest priority to the thread which is likely to have the most long latency instructions, it will eventually clog the queue and degrade the performance as [11] already specified. Thus we decide to monitor not only the numbers but also the characteristics of the instructions in the decode stage, the rename stage, and the instruction queues, that is, to keep track of the types of the instructions in stages mentioned above as well.

Appending our considerations to ICOUNT, the dynamic priority rules becomes as the following:

1. While LSQ under low usage, we let the thread which tends to have more load higher priority. This is done by monitoring the number of load instructions coming from each thread in the LSQ. Thread with more load instructions in the LSQ is considered to have more load instructions by the principle of locality.
2. If RUU usage is low, we let the thread which tends to have more floating point instructions higher priority. Similarly, we set a counter to monitor the number of floating point instructions coming from each thread in the RUU. Thread with more floating point instructions in the RUU is considered to have more floating instructions.
3. A thread that tends to have many branch instructions gets higher priority since branches pass through machine quicker and its help to reduce branch miss prediction latencies. The concept has already embedded in the ICOUNT.
4. We maintain ICOUNT rules if none of the above conditions is taken into consideration.

None of the implementation on each of these rules is more complex than ICOUNT does. As a matter of fact, they monitor the instructions in the same pipe-stages but to keep track of the instruction type.

We expect the policy above will further utilized the shared resource in simultaneous multithreaded processor and achieve higher performance.

5 Simulation and Results

In order to give a more clear view of how the parameters in our proposed policy affect the overall performance, we construct two schemes step by step according to the rules of proposed scheme.

Table 1. Simulated processor configuration

Parameter	Parameter	Value	Value
Base Fetch Policy	ICOUNT	Branch Predictor	2 level adaptive
Issue Width	8	L1 Cache Block Size	32B
Fetch Queue Size	32	ICache	32K, 2 way
Load/Store Queue Size	64	DCache	32K, 2 way
Register Update Unit Size	128	L2Cache Block Size	64B
Integer Functional Units	8	L2 Cache	512K, 4 way
Floating Point Functional Units	8		

Table 2. Latency table

Latency Type	Value (cycles)
Integer	1
FP Add	2
FP Mult	4
FP Div	12
Branch Mis-prediction Recovery	3
L1 Hit	1
L2 Hit	10
Memory Access	122

Table 3. Selected Application

Selected Applications	
Integer Based	mcf, gcc, gzip, crafty, bzip2, gap, vpr, twolf
Floating Point Based	ammp, art, mesa

In this chapter, we will first introduce our simulation framework with the simulator configurations and workloads in the first section. The two schemes we constructed will be described in the next section along with the ICOUNT, which has been chosen as our baseline scheme. The simulated results of the two schemes compared to the baseline scheme will be shown in the last section.

Table 4. Workloads of 2 threads

No	**Applications**	**No**	**Applications**
All Integer Based		**Mix of Integer and Floating Point Based**	
0	mcf, gcc	**8**	mcf, mesa
1	gzip, crafty	**9**	gzip, ammp
2	bzip2, gap	**10**	gcc, art
3	twolf, bzip2	**11**	gap, mesa
4	vpr, gap	**12**	vpr, ammp
All Floating Point Based		**13**	twolf, art
5	ammp, art		
6	mesa, art		
7	ammp, mesa		

Table 5. Workloads of 4 threads

No	**Applications**	**No**	**Applications**
All Integer Based		**Mix of Integer and Floating Point Based**	
0	mcf, gzip, crafty, twolf	**3**	mcf, bzip2, mesa, art
1	gcc, crafty, gzip, bzip2	**4**	twolf, vpr, mesa, art
2	gap, bzip2, mcf, vpr	**5**	gcc-crafty-gzip-mesa
		6	mcf-vpr-bzip2-art

Table 6. Workloads of 6 threads

No	**Applications**
All Integer Based	
0	mcf, vpr, twolf, crafty, gzip, gap
1	gap, mcf, twolf, gcc, vpr, bzip2
Mix of Integer and Floating Point Based	
2	mcf, gzip, twolf, mesa, ammp, art
3	gcc, crafty, gzip, mesa, art, ammp
4	gcc, crafty, gzip, twolf, mesa, art

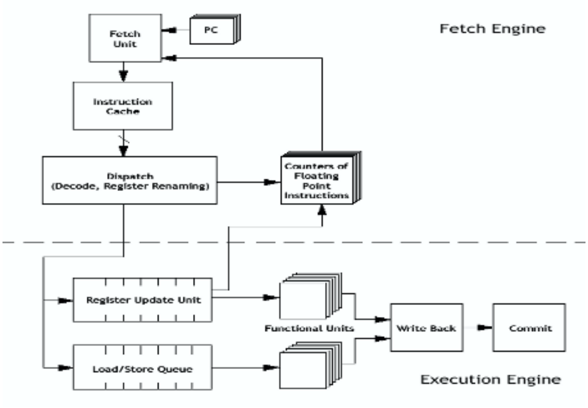


Fig. 1. The Floating Point First + ICOUNT scheme

5.1 Simulation Framework

Our simulator is derived from the SSMT simulator which originally developed by Madon [8] and further enhanced by the Vortex Project [1] at the University of Maryland. The simulator implements simultaneous multithreaded processor pipeline based on the out-of-order processor model from SimpleScalar tool set. It duplicated the SimpleScalar architecture's physical context according to the number of execution contexts to execute simultaneously. The characteristics of the simulated processor we used are given in Table 1. Table 2 shows the configurations of latencies.

We have picked up 11 applications from the SPEC CPU2000 suite to construct our workloads where 8 of them were integer based from CINT2000 suite and others were floating point based from CFP2000 suite. The applications selected are listed in Table 3. All the benchmarks were running on a GNU/Linux x86 box using reference data sets.

The workloads in our simulation are shown in Table 4 ~ 6. Table 4 lists 14 combinations of two benchmarks. Among these combinations, five of them were formed by two integer based applications; and three of them contained floating point based applications only; others were composed of an equal mix of integer and floating point based applications. Table 5 lists seven combinations of four benchmarks. Three of them were constructed by all integer applications, while others maintained an equal mix of applications from integer and floating point based. Table 6 shows five combinations of six benchmarks. Again, we had two combinations of all integer based applications and others were mixed with both kinds of applications.

5.2 Simulated Fetch Schemes

We formed two schemes from the policies proposed in section 4 and have them simulated along with the ICOUNT scheme and a random scheme to give a more clear view

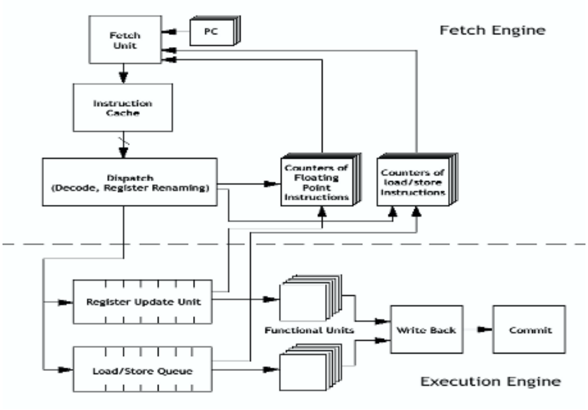


Fig. 2. The Long Latency First + ICOUNT scheme

about the impact of the parameters in the policies we proposed. The four simulated schemes are described as following:

1. Random Scheme: Random priorities are assigned to multiple threads each cycle. To avoid instructions from a single thread clog the RUU, a thread will loss its opportunity if instructions from a single thread fill 70% of the RUU. Intuitively, instructions fetched may have a fair distribution among different threads in this scheme.

2. ICOUNT Scheme: We will simulate the ICOUNT.x.8 scheme specified in [10] since we use it as our baseline scheme. It fetches up to 8 instructions from x threads each cycle, and the highest priority is given to the thread with the least instructions in the decode stage, the rename stage, and the instruction queues.

3. Floating Point First + ICOUNT Scheme: We then modify the processor and run the simulation with the following rules: while the RUU usage is lower than 50%, highest priority is given to thread tends to have more floating point instructions to execute, otherwise, we assign the priority as ICOUNT formally does. We expect the scheme to gain performance by further utilizing the RUU and floating functional units. The scheme is shown in Figure 1.

4. Long Latency First + ICOUNT Scheme: At last, we apply all the rules proposed in section 4.3 which benefits by not only utilizing the RUU but also the LSQ. It will first assign the highest priority to the threads which tens to have more loads if the usage of LSQ is lower than 60%; else it will check the usage of RUU. If the usage of RUU is lower than 50%, the highest priority is given to thread which most likely to have floating point instructions to execute. Otherwise, the ICOUNT rules are applied. The final scheme is shown in Figure 2.

5.3 Simulation Results

In this section we represent the results of simulation through Figure 3 to 7.

The weighted speedup is defined as:

IPC of proposed scheme / IPC of the base scheme

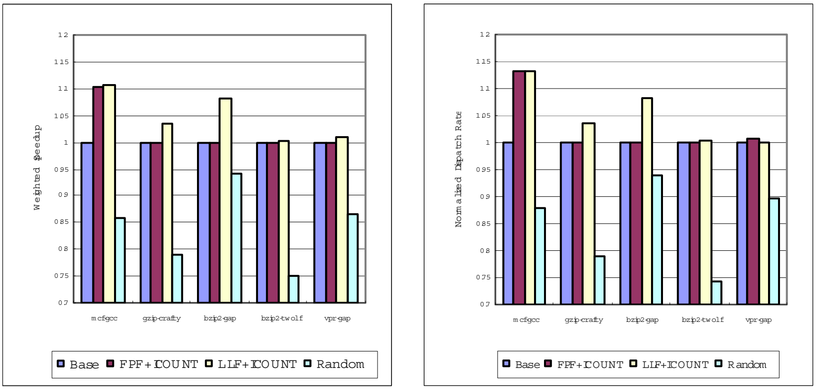


Fig. 3. Speedups and instruction dispatch rates of the workloads with two integer based threads

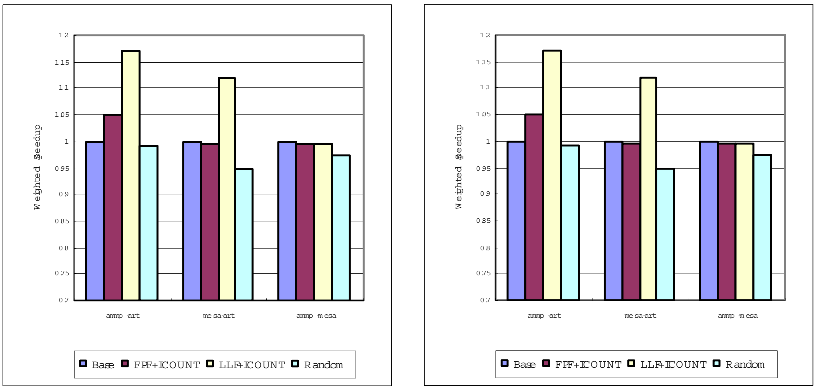


Fig. 4. Speedups and instruction dispatch rates of the workloads with two floating point based threads

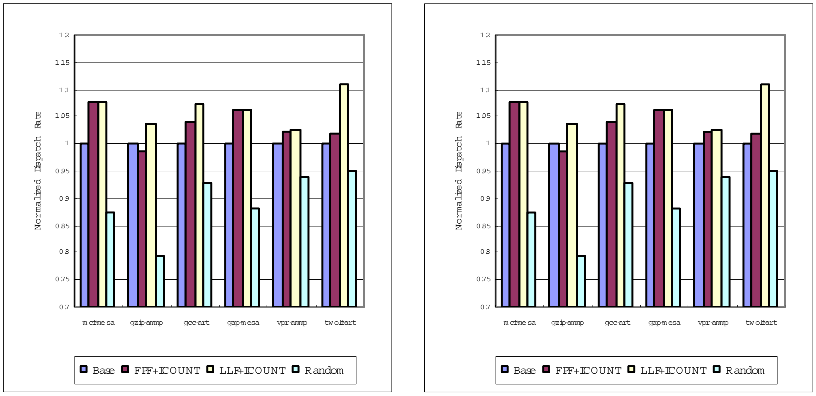


Fig. 5. Speedups and instruction dispatch rates of the workloads with equal mixed threads

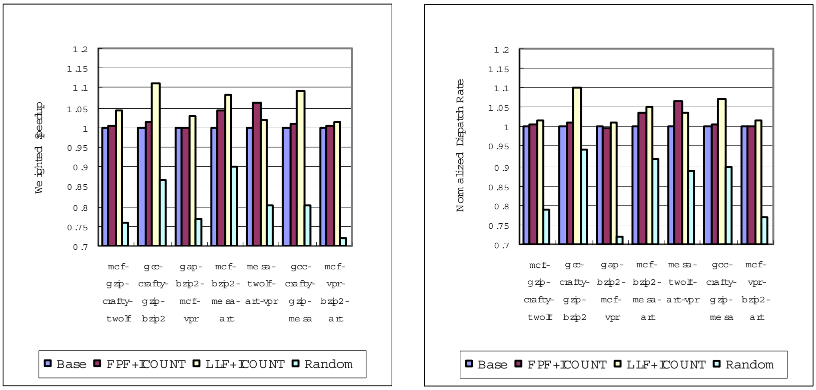


Fig. 6. Speedups and instruction dispatch rates of the four threads workloads

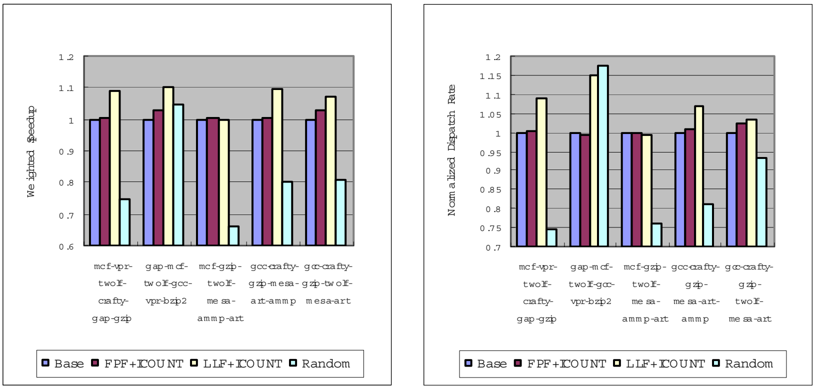


Fig. 7. Speedups and instruction dispatch rates of the six threads workloads

Figure 3 gives the speedups of our schemes relative to baseline scheme in two threads workload. All the workloads listed in this figure were integer based. Floating Point First + ICOUNT scheme didn't work in most combinations of thread, since there are either none or very few floating pointing instructions in these workloads. However, the Long Latency First + ICOUNT scheme which add the load instructions into consideration gained the overall performance.

We think the combination mcf-gcc boost the IPC because mcf is a memory- intensive application where as gcc is ILP-intensive application. ICOUNT scheme might give the higher priority to gcc in most of the execution time thus block the instructions of the mcf thread from fetching into the pipeline. Our scheme provides the ICOUNT scheme a reasonable feedback to balance the two threads.

Figure 4 gives the speedups of workloads which are all composed by floating point based threads. Floating Point First + ICOUNT scheme didn't work well in most combinations of thread. This may be because of lacking floating point functional units. Increase the floating point units may help.

The results of workloads which contain an equal mix of integer and floating point based threads are shown in Figure 5. Both the Floating Point First + ICOUNT scheme and Long Latency First + ICOUNT scheme gain improvements.

Our scheme works for four threads workload, too. Figure 5.6 gives the speedups of our schemes relative to baseline scheme in four threads workload. Most of the performances of the four configurations are increased.

Figure 7 gives the speedups of our schemes relative to baseline scheme in six threads workload.

Through all the figures also gives the improvements of overall resource usage of RUU/LSQ corresponding to the way workloads formed by representing the dispatch rate. As we can see, the shared processor resources are further utilized in most cases.

6 Conclusion

While the simultaneous multithreaded processors gain performance by sharing the processor resources dynamically to exploit thread-level parallelism along with instruction-level parallelism, it still has some potential drawbacks.

The fetch unit has been identified as one of the major bottlenecks of this architecture. Several fetch schemes were proposed by prior works to enhance the fetching efficiency. Among these schemes, ICOUNT, proposed by Tullsen et al. in which priority is assigned to a thread according to the number of instructions it has in the decode unit, register renaming unit and instruction queues were considered to be a great scheme not only the performance but also the efficiency of implementation.

The ICOUNT scheme works mainly because it favors the thread which fast moving through the pipeline, thus use the resource effectively, but this may degrade the usage of shared processor resource. We think it is better letting the thread which tends to have more long latency instructions get the priority at adequate time since long latency instructions are very likely on program's critical path.

We proposed a dynamic fetch scheme which gives the long latency bound thread higher priority while the RUU or LSQ is under low usage. Our motivation is to gain further performance by not only use the resource effectively but also by the urgency of the instructions. The proposed scheme aggressively attacks the LSQ and RUU usage which does further utilize the shared processor resources and achieve overall performance improvements. Further more, it maintains the characteristic of easy to implement.

Experiments shows that our scheme achieves an average speedup of 6% and 17% speedup in maximum with two threads workload; average speedup of 5% and 11% speedup in maximum with four threads workload; average speedup of 4% and 9% speedup in maximum with six threads workload. The resource usage is further utilized in most cases, too.

References

[1] G.. Dorai, and D. Yeung. Transparent threads: resource sharing in SMT processors for high single-thread performance. In *2002 International Conference on Parallel Architectures and Compilation Techniques (PACT'02)*, September 22 - 25, 2002

[2] S. Eggers, J. Emer, H. Levy, J. Lo, and R. Stamm, and D. Tullsen. Simultaneous multithreading: A platform for next-generation processors. *Technical Report TR-97-04-02, University of Washington, Department of Computer Science and Engineering*, April 1997.

[3] A. El-Moursy, and D. Albonesi. Front-end policies for improved issue efficiency in SMT processors. *9th International Symposium on High-Performance Computer Architecture*, pages 31-40, February 2003.

[4] H. Hirata, K. Kimura, S. Nagamine, Y. Mochizuki, A. Nishimura, Y. Nakase, and T. Nishizawa. An elementary processor architecture with simultaneous instruction issuing from multiple threads. In *19th Annual International Symposium on Computer Architecture*, pages 136-145, May 1992.

[5] P.M.W. Knijnenburg, A. Ramirez, F. Latorre, J. Larriba, and M. Valero. Branch classification to control instruction fetch in simultaneous multithreaded architectures. In *International Workshop on Innovative Architecture for Future Generation High-Performance Processors and Systems (IWIA'02)*, January 10 - 11, 2002

[6] J. Lo, S. Eggers, J. Emer, H. Levy, R. Stamm, and D. Tullsen. Converting thread-level parallelism into instruction-Level parallelism via simultaneous multithreading. In *ACM Transactions on Computer Systems*, pages 322-354, August 1997.

[7] K. Luo, M. Franklin, S. Mukherjee, and A. Sezne. Boosting SMT performance by speculation control. In *15th Proceedings of International Parallel and Distributed Processing Symposium (IPDPS)*, 2001.

[8] D. Madon, E. Sanchez, and S. Monnier, A Study of a Simultaneous Multithreaded Architecture. In *Proceedings of EuroPar'99, Toulouse, Lectures Notes in Computer Science, Volume 1685, Springer-Verlag*, pages 716-726, August 31 - September 3 1999.

[9] D. Marr, F. Binns, D. Hill, G.. Hinton, D. Koufaty, J. Miller, and M. Upton. Hyper-threading technology architecture and microarchitecture. *Intel Technology Journal*, pages 4-15, February 2002.

[10] D. Tullsen, S. Eggers, J. Emer, H. Levy, J. Lo, and R. Stamm. Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. In *23rd Annul International Symposium on Computer Architecture*, May 1996.

[11] D. Tullsen, and J. Brown. Handling long-latency loads in a simultaneous multithreading processor. In *34th Annual International Symposium on Microarchitecture*, December, 2001

[12] D. Tullsen, S. Eggers, and H. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *22nd Annul International Symposium on Computer Architecture*, June 1995.

A Novel Rename Register Architecture and Performance Analysis

Zhenyu Liu and Jiayue Qi

Institute of Microelectronics of Tsinghua University, Beijing 100084, P. R. China
{liuzhenyu, qijy}@tsinghua.edu.cn

Abstract. In today's superscalar processors, the register renaming scheme is widely used to resolve data dependence constraints. The drawback of the conventional design is that the bit-line load of the storage cell is so heavy that the access time to these storage elements is more than one cycle, impacting the IPC adversely. Moreover, in order to implement precise exception handling, the conventional allocation and recovery strategy is very complex. A novel Rename Register architecture is presented in this paper to overcome these problems. This Rename Register has such features: 1) each storage cell has just one write port, which reduces the bit line load and simplifies the circuit design, so the access time of this Rename Register could be greatly improved; 2) the allocation and recovery strategy of this Rename Register is low-complex. This feature not only simplifies the Rename Register control circuit, but also improves the exception handling speed.

1 Introduction

Contemporary superscalar microprocessors rely on aggressive execution reordering mechanisms to achieve high performance. In superscalar architecture, crucial problems include accommodating results of in-flight instructions and resolving the data dependence between instructions in program. The register renaming implementations can be divided into two categories: In the first category, such as Pentium III and AMD K5 [1] [2], the physical registers are integrated into the reorder buffer (ROB) to support register renaming; the second scheme, such as SPARC and DEC 21264 [3][4], applies one dedicated Rename Register. In these designs, because multiple function units may write their results to the same physical register entry, the physical register bitcell must have several write ports. In order to bypass the in-flight instructions' results, multiple read ports are also needed. For example [3], the storage cell of SPARC Rename Register with 4 write-ports and 10 read-ports is illustrated in figure 1. In some aggressive designs, the port number of physical register is even more [5].

The large number of ports, aside from increasing the device count linearly with the number of ports, increases the layout area for the bitcell array almost quadratically with the number of port [6], as additional ports increase each lateral dimension of a bitcell linearly. What is more, the multiple write port structure makes it impossible to

P.-C. Yew and J. Xue (Eds.): ACSAC 2004, LNCS 3189, pp. 503–514, 2004.

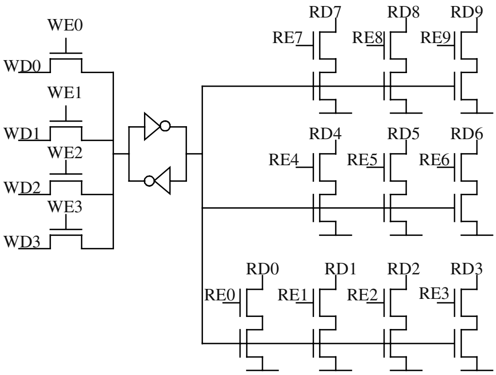


Fig. 1. 14-port data storage bit cell

reduce the bitcell load through duplicating the bitcell. Compared with ROB integrated renaming scheme, Rename Register scheme has fewer ports, but its allocation and free strategy is more complex. Two auxiliary lists, free-list and recovery-list, must be applied to maintain the precise state in case of exceptions and mispredictions. In some processors, function units (FUs) have different word width. For example, in MIPS32 architecture, the results of the arithmetic and logic unit (ALU) and the load and store unit (LSU) are 32-bit width, otherwise, the result width of the multiplication and division unit (MDU) is 64-bit. This makes the circuit design even more difficult.

A simplified Rename Register structure is proposed in this paper to overcome these draw-backs. This renaming scheme exploits the fact that the results produced by every FU are committed in program order. We divide the Rename Register into several partitions and each FU has its own private Rename Register partition. So every partition has just one write port, this character even makes it feasible to reduce the bit line load through duplicating bitcells.

Each partition is implemented as a circular FIFO queue with head and tail pointers. One entry is made at the tail of RR partition for each dispatched instruction. Instruction results are committed from the head of the RR partition to the Architecture Register File (ARF). This scheme is similar to conventional ROB. The entry width of each partition is equal to the execution unit result and depth of each partition could be optimized to save area and improve access time.

This paper is arranged as follows: In section 2, the system microarchitecture and structure of the Rename Register are presented. Dependency data bypass mechanism is explained in this section. In section 3, precise exception handling is described in details; the performance analysis in several conditions, such as level-1 D-Cache miss and level-2 D-Cache miss, is discussed in section 4 and section 5 provides conclusions.

2 Microarchitecture

In this section, the overview of the processor architecture is presented at first. Then the detailed structure and the allocation scheme of this RR are described.

2.1 Overview of the Processor Architecture

In order to analyze the function and performance of the RR, We build a RISC processor with the following features:

- MIPS32Kc instruction set compliance,
- Four function units: Jump_Branch_Unit (JBU), Arithmetic_Logic_Unit (ALU), Multiply_Divide_Unit (MDU) and Load_Store_Unit (LSU),
- MDU is non-pipelined,
- Fix map mode applied in address mapping,
- Out-of-order instruction execution.

The system block diagram is shown in figure 2. The dash line block is the Rename Register designed in this paper. The function and performance of execution units are listed in table 1 and the architecture configuration of this processor is shown in table 2.

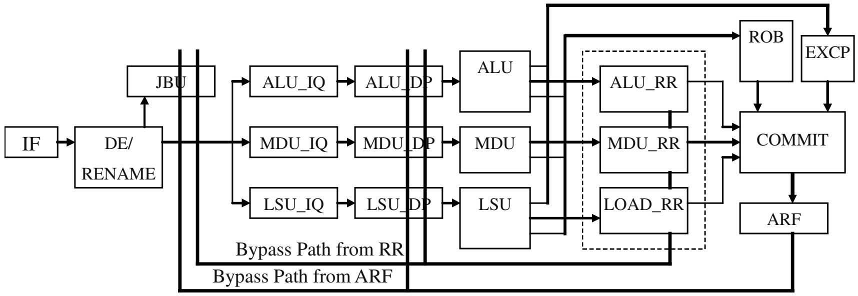


Fig. 2. The processor block diagram

Architecture registers are mapped to the Rename Register through Rename Table (RT). When an instruction enters DE/RENAME stage, it changes its destination architecture register's rename record and gets its operands' rename information from RT.

The dispatch component (ALU_DP, MDU_DP and LSU_DP) fetches the ready instruction from the head of the relative instruction queue (ALU_IQ, MDU_IQ and LSU_IQ). The operands of the dispatched instructions could be fetched from two places. If the operand is a result of an in-flight instruction and it is valid, this operand is bypassed from the RR partition. Otherwise, this operand is obtained from ARF. In the following sub-sections, we will describe how the dispatch components fetch the correct operands.

Compared with the traditional designs, we divide the Rename Register into three partitions: ALU Rename Register (ALU_RR), MDU Rename Register (MDU_RR) and LOAD Rename Register (LOAD_RR). So each partition has just one write port, which is the merit of this scheme. This architecture resolves the multiple write port problem of the traditional design. For example, if one bitcell has 12 read ports, the bit line load could be reduced through duplicating the storage cell (Fig 3). Because the bitcell area increases quadratically with port number, duplication scheme reduces the port number of every bitcell, total area of the storage array is still reduced.

Table 1. Function and operation latency of execution units

Execution Unit	Function/Latency
JBU	jump branch instructions/1 cycle
ALU	arithmetic logic instruction/1 cycle

MDU	multiply/2 cycles;
	multiply&add/2 cycles;
	divide/4 cycles

LSU	Store/1cycle Load(L1 D-Cache hit)/1 cycle Load(L1 D-Cache miss, L2 D-Cache hit)/5 cycles Load(L1-L2 D-Cache miss, L3 D-Cache hit)/10 cycles

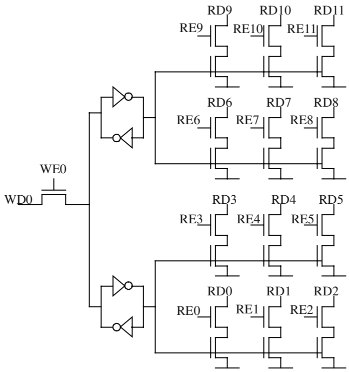


Fig. 3. Single write-port storage bit cell with duplicated storage cells

Because instructions dispatched to the same FU are committed in program order, each partition is implemented as a circular FIFO queue. The commitment component is in charge of retiring valid instructions from the heads of RR partitions. ROB just stores the crucial tags of in-flight instructions. The results of committed instructions could be obtained from the heads of RR partitions. In the following subsections, we will illustrate the structure of IQ, RR and ROB and explain the principle of how to implement the dependence data bypass.

Table 2. Architectural configuration of the processor

parameter	configuration
Machine wide	1-wide issue, 2-wide commit
ALU_RR	32 entry
MDU_RR	16 entry
LOAD_RR	32 entry
ROB	80 entry
ALU_IQ	32 entry
MDU_IQ	16 entry
LS_IQ	32 entry

2.2 Details of the Components

In order to implement this Rename Register design, other components, such as IQ, RT, ROB and EXCP, must be devised deliberately. We also show the structure of these components and explain how they work.

Rename Register Partition. Each RR partition is a circular FIFO. It has two pointers: head pointer indicates the first entry to be committed and tail pointer indicates the first empty entry to be allocated. Two flags, “empty” and “full”, are used to present the status of the RR partition.

Every entry includes two fields, data field and valid field. The word width of the data field is equal to the result of the execution unit. For ALU and LSU, their RR partitions are 32-bit width. For MDU, because its result is 64 bits, MDU_RR is designed to 64-bit width. The depth of each partition can be optimized through usage statistic.

Once one instruction enters DE/RENAME stage, it is assigned one entry of RR partitions depending on its execution unit. For example, ALU instructions get entries from ALU_RR and MDU_RR is dedicated to MDU instructions. If the required partition is in “full” status, this instruction must be stalled.

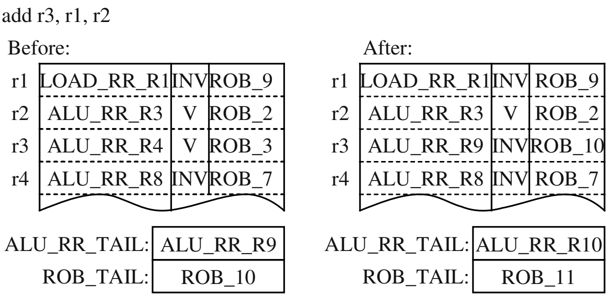
The valid field is 1bit width. The valid bit of one allocated entry is set when the owner instruction writes its in-flight result to this entry. Now this entry’s data can be bypassed to other consumer instructions in pipeline. So, each RR partition must have multiple read ports for data forwarding. When one instruction is committed, its RR partition entry is freed and the valid bit is reset.

The advantages of this scheme include: 1) the word width changeable feature simplifies circuit complexity; 2) the depth optimization can efficiently reduce the scale of RR partitions.

Rename Table Structure. In MIPS32Kc architecture, ARF includes Register File, Hi Register, Lo Register and CP0. Because CP0 is seldom written, scoreboard strategy is applied to it. So RT has thirty-four entries, where thirty-two entries for ARF (R0-R31), one for Hi and one for Lo. Each entry includes three fields: (1) a valid bit to indicate if the data in ARF is valid; (2) a map field (7 bits) to map this logic register to a physical register in RR, the most significant two bits are used to address the three partitions (00 indicates ALU_RR; 01 indicates LOAD_RR; 10 indicates MDU_RR); (3) a tag field (7 bits) to record the ROB tag of the last instruction that modifies this register.

When one instruction enters DE stage, it gets its operands map information from RT and modifies the RT entry of its destination register. At the same cycle, this instruction is assigned one available entry from the tail of ROB and one available entry from the tail of the RR partition. The ROB tag of this instruction is stored in the ROB tag field and the physical address of the RR entry is stored in the map field.

An example in figure 4 illustrates the procedure of physical register renaming. The logical architect registers are denoted with lower case letters and the physical registers use upper case. In the before side of the figure, the mapping table shows that register r1 maps to physical register LOAD_RR_R1 and its valid flag is reset. So r1 is mapped to LOAD_RR_R1. Because the other operand r2 is valid, this operand is still mapped



add r3, r1, r2

Before:

r1	LOAD_RR_R1	INV	ROB_9
r2	ALU_RR_R3	V	ROB_2
r3	ALU_RR_R4	V	ROB_3
r4	ALU_RR_R8	INV	ROB_7

ALU_RR_TAIL:	ALU_RR_R9
ROB_TAIL:	ROB_10

After:

r1	LOAD_RR_R1	INV	ROB_9
r2	ALU_RR_R3	V	ROB_2
r3	ALU_RR_R9	INV	ROB_10
r4	ALU_RR_R8	INV	ROB_7

ALU_RR_TAIL:	ALU_RR_R10
ROB_TAIL:	ROB_11

Fig. 4. Example of register renaming. Logical registers are shown in lowercase and physical registers are in upper case

to r2. During renaming, the ‘add’ instruction’s source register r1 is replaced with LOAD_RR_R1, the physical register where a value will be placed by a preceding instruction. The destination register r3 is renamed to the first free physical register ALU_RR_R9, and this renaming is recorded in the mapping table. Before this ‘add’ instruction is committed, any subsequent instruction that reads the value produced by this ‘add’ will have its source register mapped to ALU_RR_R9, so that it gets the correct value.

Instruction Queue Structure. Each FU has its own instruction queue and instructions are steered to the proper instruction queue depend on their execution units. In order to reduce power dissipation and circuit complexity, the issue strategy is circular FIFO queue, which means that the dispatch logic fetches the ready instruction from the head of each instruction queue and entries are allocated from the tail.

Every IQ entry must store the operand information, which is used during instruction dispatch stage. The information includes: 1) operation code; 2) ARF addresses of source operands; 3) RR addresses of source operands; 4) valid bits of source operands 5) ROB tags of the operands; 6) the destination ARF address; 7) the destination RR address; 8) the ROB tag of the instruction; 9) ALU and LSU can cause exception, so ALU_IQ and LSU_IQ should store the instruction PC for exception handling. The formation of ALU_IQ entry is shown in figure 5.

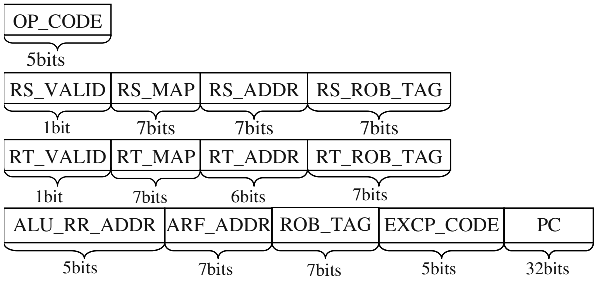


Fig. 5. ALU_IQ entry formation

IQ is also responsible for updating source dependence of instructions in the IQ waiting for their source operands to become available. Every time an instruction is committed, the ROB tag associated with this instruction is broadcast to all instructions in IQs. Each instruction then compares the tag with the tags of its source operands. If there is a match, the operand is marked available by setting its valid flag (RS_VALID or RT_VALID). This means that this source operand is available and should be fetch from ARF other than from RR partitions. Figure 6 illustrates the dynamic update logic. TAG1 and TAG2 represent the ROB tags of the two committed instructions. Reference [7] provides some methods to reduce the energy consumption of issue logic, such as disabling the wake-up for empty and ready entries and dynamic resizing of IQ. After applying these approaches, the IQ power consumption could be greatly reduced.

Dependency Data Bypass Mechanism. There exits two status of one source operand. First, the producer instruction of this source has been committed. In this case, the valid flag in IQ of this operand must have been set and this data should be fetched from ARF. Second, the producer instruction is still in-flight. Now, the consumer should check the RR partition entry allocated for the producer, if this producer has generated the result, it can be bypassed from RR, otherwise the consumer must wait until this source is generated. When all source operands of one instruction in the head entry of one IQ are available, this instruction is ready to be dispatched.

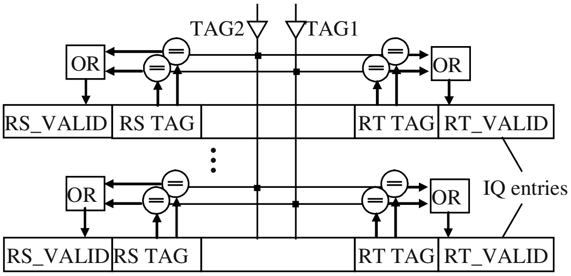


Fig. 6. IQ dynamic update logic

ROB Structure. The entry in the conventional ROB [6] includes at least following fields: (1) a result field to hold the value generated by the instruction that targets a register (32bits); (2) a bit to indicate if the result field is valid (1bit); (3) the address of the instruction ("PC value" 32bit); (4) exception codes (5 bits) and (5) architectural register id (7 bits, used for updating the architectural register within the ARF at the time of committing the instruction). If the depth of the ROB is D, the whole scale of the ROB is (32+1+32+5+7)×D bits, which consumes a non-trivial fraction of the total chip area and power. In the architecture presented in this paper, ROB scale is greatly reduced. At first, the in-flight instruction results are stored in the Rename Register; second, a dedicated exception component is applied to storage the exception message of the first exception instruction. So ROB has just these fields: (a) the destination architectural register ID (7 bits); (b) Rename Register partition ID (2 bits); (c) one bit to indicate the validity of the entry. The ROB scale is reduced to (7+2+1) ×D bits. The depth of ROB is much deeper than other storage component. For example, in our design, the depth of ROB is 80. The area reduction of ROB causes the shortage of the bit line and word line of storage array. This optimization not only speeds up the access time of ROB but also lowers the power dissipation.

Commitment fetch the valid instruction from the head of ROB and the result of this instruction is fetched from the head of the RR partition denoted by the RR partition ID. The destination address is denoted by the destination address field of this ROB entry, where [0, 31] indicates architecture register file, [32, 63] indicates CP0 register, [64] indicates HI, [65] indicates LO, [66] indicates HILO and [67] indicates memory.

It is obviously that the ROB and all Rename Register partitions are circular FIFOs. This feature simplifies the management of RR and ROB, especially when the exception occurs.

3 Precise Exception Handling Mechanism

In this design, a dedicated exception component is applied to implement precise exception processing. As described in the following sub sections, this approach greatly reduces the scale and complexity of ROB. In section 4, the synthesized result shows that 80 entries ROB is reduced to 9% of total area, which dose not include caches. Compared with our design, traditional design ROB occupies no trivial area. For example, HP8000 ROB consumes 20% of whole die area [6].

3.1 Exception Component

In this microarchitecture, a dedicated exception component is applied to store the first exception instruction according to the program order. In fact, just the information of first exception is useful, because once this instruction is committed, it cancels the following instructions in pipeline. Exception component has four fields:1) a valid bit to indicate if the recorded exception information is valid; 2)a ROB tag (7bits) to store the ROB tag of the exception instruction; 3)a code field(5bit) to indicate the exception cause; 4) a PC field(32bits) to store the program counter of the exception instruction. ALU and LSU can both generate exception, so exception component has two groups of write ports. Figure 7 summarizes the port requirement.

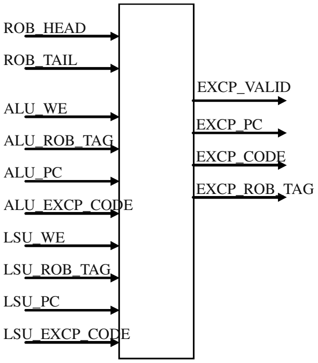


Fig. 7. Port requirement of exception component

Through comparing the ROB tags of the being written exception instructions and the stored exception instruction based upon the head and tail pointer of ROB, the exception component decides which is the first exception instruction and stores its information, such as PC, ROB tag and exception code.

3.2 Precise Exception Handling

For precise exception processing [10], Rename Register strategy is more complex than ROB strategy. Conventional Rename Register applies RAM scheme [8][9] or CAM scheme [3][4][8]. In RAM scheme, the map table is a RAM where each entry contains the physical register address that is mapped to the logical register. A shift register, present in every entry, is used for checkpointing old mappings. The width of individual entries is a function of the number of checkpoints because this number determines the length of the shift register in each entry. The CAM scheme uses two lists, free list and recovery list, to implement precise exception. When one exception instruction enters the commitment, the old mapping is recovered from the recovery list, this operation always takes more than one clock cycle.

In our design, the recovery operation is similar to ROB: When the valid of the exception component is set and the ROB tag stored in the exception component is equal to ROB head pointer, which means one exception instruction is being in commitment stage. Such operations should be taken to keep the precise status: 1) all entries in RT are set valid, that means all data stored in ARF are valid; 2) All RR Partitions and ROB are set empty, which is done by setting all entries invalid and equalizing the head and tail pointer; 3) The precise information, such as PC and exception code, are stored in CP0. In this way, the interrupted process could be resumed after the exception processing. These operations can be completed in one clock cycle. What is more, there is no auxiliary complex logic to realize this strategy.

4 Experimental Results

The performance of a microarchitecture depends on two aspects:1) instructions issued per-cycle (IPC); 2)critical path delay [8]. In this paper, we first compare the ordinary in-order architecture with this design. Next, we use TMSC0.18 standard library to synthesis this design with Synopsys DC and give the critical path delay and area overhead. The synthesized result is a useful guide for the circuit design.

We choose some benchmarks to test the performance of our design and compare them with the in-order architecture in different circumstances. During these simulations, we also can get the usage statistic of the RR partitions. Five benchmarks are applied, which include FFT, Taxis, Vector Multiplication, Matrix Multiplication and Matrix Absolute Subtraction. These algorithms are widely used in digital signal processing. Table 3 shows the percentage of instructions requiring different function units in these test vectors.

The IPC of the design applying the novel Rename Register and the conventional in-order with bypass architecture is compared in three cache statuses: (1) L1 cache hit; (2) L1 cache miss, L2 cache hit; (3) L1 and L2 cache miss, L3 cache hit. The

Table 3. Function unit utilization

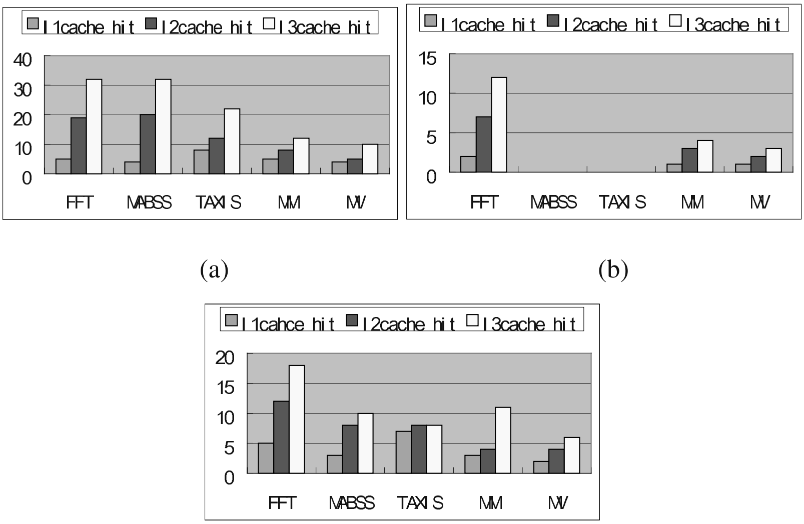
Programs	ALU Inst.	MDU Inst.	LSU Inst.
FFT	48.3%	15.9%	35.8%
Taxis	77.7%	0%	22.3%
VM	72.6%	9.1%	18.3%
MM	61.4	16%	22.6%
MABS	80.6%	0%	19.4%

effect generated by cache miss is the problem that we are interested, because cache miss not only reduces the performance but also affects the RR partition usage. The performance comparison between these two architectures is shown in table 4.

Table 4. Performance comparison of this architecture and in-order architecture

Programs	this architecture(IPC)			in-order architecture(IPC)		
	L1-hit	L2-hit	L3-hit	L1-hit	L2-hit	L3-hit
FFT	1.000	1.000	1.000	0.863	0.801	0.701
Taxis	1.000	1.000	1.000	1.000	0.979	0.955
VM	1.000	1.000	0.997	0.913	0.777	0.655
MM	1.000	1.000	0.998	0.862	0.813	0.759
MABS	1.000	0.929	0.795	0.933	0.789	0.642

Applying the Rename Register, the performance is improved 10%-20%, especially when multi-cycle instructions are included, such as MDU instructions or L1-L2 cache miss occurs. Diagrams of RR partition usage under these test vectors are illustrated in figure 8.



(a) (b)

(c)

Fig. 8. Rename Register partition usage diagrams. (a) IU_RR usage diagram (b) MDU_RR usage diagram (c) LOAD_RR usage diagram

It is obviously that in different system configuration the partition usage status changes greatly. For example, in FFT benchmark, when L1-cahche hit, the maxim ALU_RR usage is 5 entries, otherwise, when L1-L2 cache miss and L3-cache hit, the maxim usage is 32 entries. This feature provides a useful way to optimize the design. In today's processor, because IO speed is much lower than internal logic, when cache-miss occurs, it must take tens of internal clock cycle to fill the missed cache line. The stall generated in this condition can not be avoided with reasonable RR hardware overhead. In these designs, the duty of RR is resolve the stall cause by data dependence and multi-cycle components, such as MDU operation, FPU operation or L1-L2 cache miss. In many embedded processors, there are no L2 and L3 caches. In these processors, the scale of the Rename Register can be greatly reduced and IPC will not be affected.

In order to provide useful data for hardware design, this design is synthesized with TMSC 0.18um standard cell library to analyze the design area and work clock speed. Besides cache (I-Cache and D-Cache), the total design area is 267926 gates. The area percentage of components in this processor is shown in table 5.

Table 5. Components size

components	area (gates)	area percentage
Rename Register & Rename Table	78607	29%
Register File	30680	11%
ROB	24631	9%
ALU_RSVST	36799	13%
LSU_RSVST	36014	13%
MDU_RSVST	15677	6%
ALU	6641	2.5%
MDU	16050	6%
LSU	9197	3.4
others	92237	7%

In our design, we combine the Architecture Register with Rename Register into one cluster, which is donated Regset. Function units get their operands from Regset, so the access time of Regset is very important. After synthesized with TMSC 0.18um standard cell library, in typical condition, the critical access time of the Rename Register is 1.61ns.

5 Conclusion

Through the optimized partition scheme, the storage cell in Rename Register partitions has just one write port. This architecture will bring following merits in the Rename Register circuit design: 1) Without multiple write ports, the circuit is simplified and the die area is greatly reduced; 2) the bit line load of the storage cell can be reduced, so the read access time is improved; 3) the scale of every Rename Register partition can be optimized flexibly depending on the usage; 4) the control logic of this Rename Register is as simple as ROB scheme. A synthesized result of this processor presented in this paper also provides some useful guidance for the circuit design.

Acknowledgement. Authors would like to thank Intel Corporation for its significant contribution to this research.

References

1. Case, B.: Intel Reveals Pentium Implementation Details. Microprocessor Report, Vol. 5, No. 23 (1993) 9-17
2. Slater, M.: AMD's K5 Designed to Outrun Pentium. Microprocessor Report, Vol. 8, No. 4 (1994) 1-7
3. Asato C.: A 14-Port 3.8ns 116-Word 64b Read-Renaming Register File. IEEE Journal of Solid-State Circuits, Vol. 30, No. 11 (1995) 1254-1258
4. Kessler, R.E.: The Alpha 21264 microprocessor. IEEE Journal of Micro, Vol. 19, No.2 (1999) 24-36
5. Jolly, R.D.: A 9-ns 1.4-Gigabyte/s 17-ported CMOS register file. IEEE Journal of Solid-State Circuits, Vol. 26, No. 10 (1991) 1407-1412
6. Kucuk, G., Ponomarev, D., Ghose, K.: Low-Complexity Reorder Buffer Architecture. Proceedings of the 16th International Conference on Supercomputing (2002) 57-66
7. Folegnani, D., Gonzalez, A.: Energy-effective issue logic. Proceedings of 28th Annual International Symposium on Computer Architecture (2001) 230-239
8. Palacharla, S.: Complexity effective superscalar processor. PhD Thesis, University of Winsconsin, Madison (1998)
9. Yeager, K.C.: The MIPS R10000 superscalar microprocessor. IEEE Journal of Micro, Vol. 16, No. 2 (1996) 28-41
10. Wang, C.-J., Emnett, F.: Implementing precise interruptions in pipelined RISC Processors. IEEE Journal of Micro, Vol. 13, No. 4 (1993) 36-41

A New Hierarchy Cache Scheme Using RAM and Pagefile

Rui-fang Liu[1], Change-sheng Xie[1], Zhi-hu Tan[1], and Qing Yang[2]

[1] National Storage System Lab.,
Huazhong University of Science and Technology, Wuhan, Hubei, China
{relyfang, csxie, stan}@hust.edu.cn
[2] High Performance Computing Lab.,
University of Rhode Island, Kingston, RI 02881, U.S.A.
qyang@ele.uri.edu

Abstract. One newly designed hierarchical cache scheme is presented in this article. It is a two-level cache architecture using a RAM of a few megabytes and a large pagefile. Majority of cached data is in the pagefile that is nonvolatile and has better IO performance than that of normal data disks because of different data sizes and different access methods used. The RAM cache collects small writes first and then transfers them to the pagefile sequentially in large sizes. When the system is idle, data will be destaged from the pagefile to data disks. We have implemented the hierarchical cache as a filter driver that can be loaded onto the current Windows 2000/Windows XP operating system transparently. Benchmark test results show that the cache system can improve IO performance dramatically for small writes.

1 Introduction

Rapid advances in semiconductor technology have dramatically increased the speed gap between RAM and disks because of the mechanical nature of magnetic disks [1]. Magnetic disks must rotate the spindle and seek for the right track for every access [2]. As a result, disks have become the major performance bottleneck of a computer system. Extensive research has been reported in the literature. Existing studies on improving disk performance can be classified into two categories: improving the disk subsystem architecture and improving the software that control and manage the disk system [3].

RAID (Redundant Array of Independent Disks) is the most important architectural advance in disks in recent two decades [4]. The wide use of RAID in the computer industry has shown that RAID is a cost-effective way to obtain high performance and high reliability. The most popular RAID configuration is RAID5 in practical applications. However, RAID5 performance suffers from "small write" penalty because for every small write, 4 disk operations are required to read old data/parity and write new data/parity [5]. To mitigate the penalty, Stodolsky et al proposed a very interesting solution to the small write problem by means of adding a log buffer in controller's memory for parity logging [6]. They have shown that the solution can

P.-C. Yew and J. Xue (Eds.): ACSAC 2004, LNCS 3189, pp. 515–526, 2004.

eliminate performance penalty caused by the RAID architectures for small writes with minimum overhead. Another interesting study was done by K.H. Yueng and T.S. Yum who presented a dynamic parity disk array for engineering database systems [7].

Besides high reliability, the primary objective of the RAID architecture is to improve throughput by means of parallelism rather than reducing access latency. In office/ engineering environment, workloads are usually random and scattered with moderate average throughput. For such workloads, performance enhancement due to RAID is limited. In addition, in today's commercial computing environment, write traffic has dominated disk traffic and may potentially become a system bottleneck. There has been a great amount of efforts to improve such write performance in file systems that control and manage disks. Log-structured file systems (LFS) can provide efficient writing even for small files [8]. LFS file systems have been implemented in prevalent operation systems. NTFS improves write performance by using a write-back caching strategy [9] that writes modifications to the cache and flushes the cache to disk as a background thread. It also logs every transaction as a log record in a log file and the file system check is based on the log record. In LINUX, ext3 and reiser file systems are all supported [10]. Ext3 and reiser are used as default file system for RedHat and SuSE LINUX distribution respectively. They all provide metadata journaling. With metadata journaling, the file system metadata is going to be rock solid, and exhaustive fsck is not needed. According to the logged metadata, fsck can finish in a few seconds without scanning the entire file system.

Caching is the main mechanism for reducing response time [1] and large RAM caches are generally used to speed up disk accesses. Such caches more effectively improve read performance than write performance, since write requests must be frequently written into disks to protect them from data loss or damage due to system failures. NVRAM (Non-Volatile RAM) caches can be used to improve write performance, but large NVRAM caches are prohibitively expensive for many applications. EMC Symmetrix 8000 has caches of 2GB to 32GB and claims to have 90% to 95% read hit rates for the largest cache size [11]. The large caches exploit spatial and temporal locality to reduce accesses to the disks, but they increase the cost of system.

This paper presents a design and implementation of an efficient and inexpensive hierarchical cache for improving disk IO performance on Windows 2000/Windows XP. The design involves a new hierarchy cache using a RAM and one pagefile. While our design is based on the DCD (Disk Caching Disk) architecture [12] proposed by Hu and Yang, we proposed a new way of implementing the cache disk using pagefile instead of physical disk or logical partition giving rise to greater flexibility and ease to install and use. Users can install our hierarchical cache at any time without the need of a new physical disk or doing a disk partition. The new cache structure can potentially improve disk write performance by 2 orders of magnitudes in the office/engineering environment. The new hierarchy cache converts multiple small writes into a large write thereby reducing the total access times. Measured performance results show that the server that loads the hierarchy cache driver runs significantly faster than the traditional system. For small and bursting writes, the hierarchy cache driver can improve synchronous writes by a factor of 6.5 in terms of response time seen by users. It can reduce the mail server response time by a factor of 3.6 in heavy workload cases. The filter driver is a WDM (Windows Driver Model) filter driver. It can be inserted into the disk driver stack transparently without requiring any changes to the current operation systems.

The paper is organized as follows. The next section presents the overview of the hierarchy cache and the system architecture, followed by the detailed descriptions of the design and implementation of the hierarchy cache driver in Section 3. Section 4 discusses the benchmark programs and the measured results. We conclude the paper in Section 5.

2 Theoretical Background

For disk accesses, there are 4 components that contribute to the total access time: controller overhead, seek time, rotational latency and data read/write time. The read/write time is a very small fraction even page-sized transfers often take less than 5% of the total access time. Disks spend most of their time waiting for seek and rotation. Therefore, amount of data accessed for each disk operation affects greatly the disk performance. The larger the data size, the better the I/O performance will be since more data are transferred for each time consuming seek and rotation.

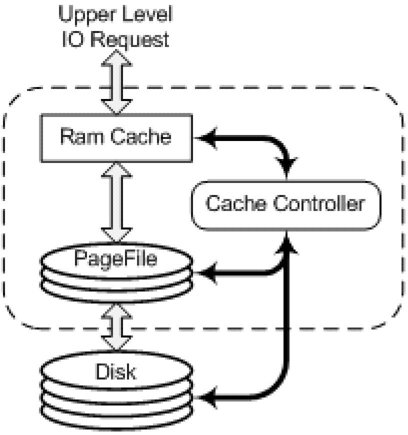


Fig. 1. Hierarchy cache architecture

The fundamental idea of the hierarchy cache is to use a pagefile, as an extension of a small faster RAM buffer on top of data disks where a normal file system exists. The RAM buffer and the pagefile together form a two-level cache to buffer write data. Small writes are first collected in the small RAM buffer. When the RAM buffer becomes full or the destaging is triggered, the hierarchy cache writes data in the RAM buffer, in multiple large and sequential data transfer, into the pagefile. These large writes finish quickly since they require only one seek instead of tens of seeks. As a result, the RAM buffer is very quickly made available again to buffer new incoming IO request packets. The hierarchy cache exploits the performance differences in different ways of disk accesses. The two-level hierarchy cache appears to the host as a large virtual NVRAM cache with a size close to the size of the pagefile. When data disks are idle or less busy, the hierarchy cache will destage from the pagefile to data disks. The destaging overhead is quite low because most data in the pagefile are short-lived and are quickly overwritten therefore requiring no destaging at all. The pagefile is much larger than the RAM cache. So the hierarchy cache can filter many IO requests, and take full advantage of spatial and temporal locality of disk accesses. The pagefile is non-volatile and hence highly reliable, too.

The hierarchical cache presented here extends the concept of DCD by using a pagefile as the cache disk as opposed to using a physical or logical partition as the cache disk. The advantage of using the page file instead of a partition is its flexibility and ease of use. Users can install the hierarchical cache any time without using a new physical disk or doing disk partitions that may destroy the current data in the disks. A pagefile has been employed to implement virtual memories in the past. Windows utilizes a pagefile to expand the physical memory and LINUX uses it for a swap partition where paging and swapping take place. In our implementation, we use the pagefile to expand the RAM cache. When we allocate space for the pagefile, we make sure that its space is physically continuous. After we intercept an IRP (IO request packet) from a user, we can keep the data in the pagefile, and we can exploit the speed difference between large sequential access and small random access of disks. By placing the pagefile in the current data partition, we do not need change the partition layout of the current system. The hierarchy cache filter driver will automatically manage the two levels of the hierarchy cache represented by the pagefile and RAM cache.

3 Design and Implementation

This section describes the key data structures and algorithms used to implement our hierarchical cache driver. As described above, we use a pagefile as the equivalent of cache disk in DCD to avoid modifying the current system partition layout. The filter driver can be inserted into the disk driver stack. It is a high level filter driver that typically provides added-value features for disks. The filter driver is a kernel-mode WDM driver and is source-code compatible with all Microsoft Windows operating systems. The filter driver can filter all the IRP targeted to data disks and reroute the IRP to the RAM cache or the pagefile according to the cache algorithm. There are many layered device drivers to finish a read/write operation. These drivers form a driver stack, and the IO manager of Windows will pass the IRP in the stacked drivers. With the help of the stack driver architecture, we can easily intercept the IRP. In order to develop the hierarchy cache driver, we use the Win2000 DDK (Device Driver Kit) and Numega Driver Suite. Numega Driver Suite is an object-oriented development tool for windows device drivers which can simplify the development of filter drivers so that the developers can concentrate on the key functionalities.

3.1 KHcDataDevice

KHcDataDevice is the son class of KFilterdevice that represents the virtual device class of the filter driver. In the driver initialization, we create an instance of the class and specify the target device when we initialize the instance. After that, we insert our driver into the target device driver stack. Our filter driver handles all the target device's standard dispatch routines, such as read, write, close, and create. After updating the new driver stack for the target device, the IRP is dispatched through the hierarchical cache driver by IO manager automatically. The main definitions of the class are as follows:

```
class DcdDataDevice : public KFilterDevice
{
public:
        DcdDataDevice(PCWSTR pwsTargetDevice);
        virtual NTSTATUS OnIrpComplete(KIrp I, PVOID Context);
        virtual NTSTATUS Write(KIrp I);
        ......
        PageFile *m_pPageFile;
        KList<HC_RAM_SECTOR> m_RamSector[HC_SECTOR_HASHSIZE];
        KList<HC_HISTORY_SLOT> m_HistorySector;
        ULONG m_WriteInLastPos;
        ULONG m_ReadInLastPos;
        NTSTATUS R2pfDestaging();
        NTSTATUS Pf2dDestaging();
        .....
};
```

We can use m_pPageFile to access the pagefile and m_RamSector to access the RAM. The member m_WriteInLastPos and m_ReadInLastPos record the read and write times in last statistical interval. With the help of them, we can judge if the system is idle or less busy. We can call the member function R2pfDestaging to destage data from the RAM cache to the pagefile. In the idle time, we should execute Pf2dDestaging to transfer data from the pagefile to data disks. m_HistorySecotro is used to record all the sectors in the pagefile. We can search all the sectors in the pagefile through the list, when we handle the read/write operations.

3.2 RAM Image

The RAM image plays the role of the RAM cache in our hierarchical cache. It is a set of contiguous memory locations reserved and allocated from the system nonpaged pool by using ExAllocatePool upon initialization of the hierarchical cache driver. m_RamSector is the hash table for the RAM. The disk is block device that is accessed in unit of sectors (512 bytes). Therefore, we divide the RAM image into sectors with the size of 512 bytes. After the filter driver intercepts a small write, it will allocate proper sectors according to the request size. Then every sector should be inserted into the proper hash table according to the sector offset. The hash table length is dynamic and there is no need to worry about collisions and replacements. We keep the hash table in sorted order when we insert a node into the list. In the mean time, we add two pointers to the other lists for every sector node to keep the continuity of sectors. Then every node is in the "+" lists. We introduce dynamic hash lists and serial sequential list, so we can position the first sector of the IO request packet with the help of the dynamic hash lists quickly, and we can get the continuous sectors according to the serial sequential list.

3.3 PageFile

In our implementation, the pagefile plays the role of cache disk in the DCD architecture [2] [3]. One pagefile can serve multiple data disks. We record the index information in the superblock, and we divide the pagefile into zones. Every zone is responsible for a partition. When data are destaged from the RAM cache to the pagefile, we write the pagefile in large writes (one segment: 64K). So we organize the pagefile using segments. Every segment has two parts. The first 1K keeps the index of the metadata of 126 sectors, and the other keeps the metadata contents. The detailed organization of the pagefile is shown in Figure 2.

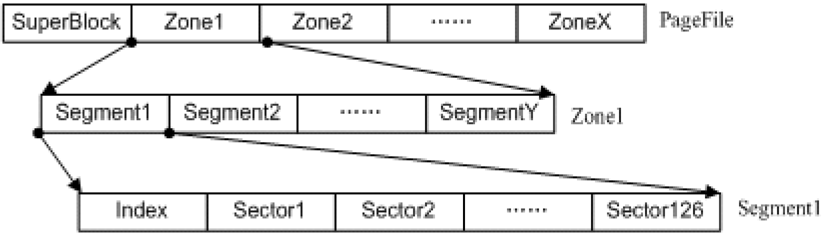


Fig. 2. Organization of pagefile

3.4 Basic Operations

3.4.1 Write Operations

When the driver filters a write request, it will retrieve the request size first. If it's a large write, the filter will pass through the request to the target disk and register a completion routine. When the request comes back, the completion routine will be executed. In the routine, we scan the RAM cache for every sector first, and we scan the pagefile later. If we find one sector, we must mark the sector stale. If the request is a small write, the filter driver will judge if it has enough spare sectors in the RAM cache. When the RAM cache is not large enough to hold the small write, it will start the destaging process to move data from RAM cache to the pagefile in large write. For every sector in the small write, we must record it in the hash tables. If the sector is already in the RAM cache, we need only update the metadata. In the process of insertion, we must keep the table in sorted order. After inserting the sector in the hash tables, the history list is also searched for the sector and marked stale if found.

3.4.2 Read Operations

When we intercept a read operation, the requested data may reside in one of multiple of three different places: the RAM cache, the pagefile and the data disk. For every sector in the read request packet, we search it in the RAM cache first. If we cannot find, we continue to search it in the pagefile according to the history list. If failed in the pagefile, the data will be read from the data disk. If the requested data is in the pagefile, we must read the entire segment where the sector is located, copy the needed sector from the segment, and update the hash tables to indicate that the segment is now in the RAM cache. The segment will stay in the RAM for future accesses.

3.4.3 Destage

The hierarchical cache consists of a RAM cache and the pagefile of top of data disks. There are two different destaging operations: destaging data from the RAM cache to the pagefile and destaging data from the pagefile to the data disk. The RAM cache is a few megabytes in size, and it is used to collect random small writes. When the RAM cache is full, we replace the sectors according to the LRU algorithm. The sectors to be destaged are combined to form a large chunk of data to be written into the page file sequentially. At the same time, we record every sector that is destaged in the history list. If the system is idle, the destaging thread will start after every 50 seconds to update the data in the RAM cache. The process is same as the forced destaging process. Destaging from the pagefile to data disks will happen in the system thread context. The thread priority is lower. When the system is idle and the filter driver finds the pagefile usage is higher than a predefined high water mark, the thread will start the destaging process. The process will read the segments from the pagefile according to the FIFO algorithm in large writes. In the previous destaging, we have combined the continuous sectors. We will build new IRP for every continuous block in the segments. And then we dispatch the IRP to the lower target device. When the pagefile usage is lower than the specified low water mark, the thread will stop destaging. We must update the history list whenever we do the destaging.

4 Performance Evaluation

4.1 Experimental Setup

To evaluate the performance of the hierarchical cache, and to give a realistic performance evaluation and comparison, we use different real world benchmarks to evaluate the effects of the hierarchical cache on an IO subsystem. We measure the performance results on a same server with or without the hierarchical cache filter driver for comparison purpose. All the tests are performed on a Tiger 7200 PC server with Pentium4 1.4G processor running Windows 2000 advanced server with service pack 1. The server is configured with 256M-SDRAM and high-performance disk with capacity of 60G. The disk is divided into three partitions. The first is the system partition, and we deploy the Exchange 2000 mail server in the second partition. And we place the data and the pagefile in the third partition. Two megabytes of system RAM is allocated to play the role of the RAM cache. Because of the limitation of the operating system, the maximum IO request packet size is 64K bytes, and the minimum IRP size is 512 bytes. In the following tests, the block size varies between 512 bytes and 64K bytes.

4.2 Benchmarks and Results

4.2.1 NTIOGEN

NTIOGEN is ported from a popular benchmark of UNIX. The input parameters of NTIOGEN are read/write percentage, IO block size, number of processes and random/sequential percentage. The output parameters are average response time, IOs per second, throughput. In the tests, we vary the block size while keeping other input parameters unchanged to observe the relationship between the average response time and the block size.

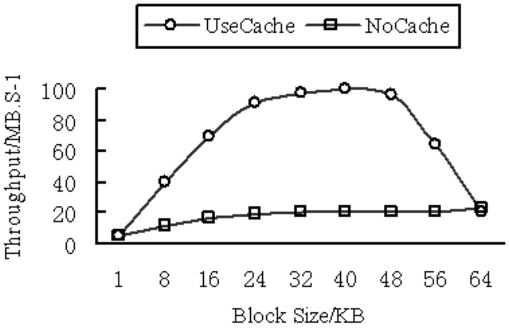


Fig. 3. Throughput comparison with different block size

NTIOGEN test results are shown in Figure 3 where the through is plotted against block size. The workload in the tests is close to the realistic workload of a working server. When there is a request, it is usually accompanied by a cluster of requests in a short time frame. In addition, there is usually a relatively long period of idle time between two consecutive request bursts. In the tests, the read/write ratio is 0/100, and the random/sequential ratio is 50/50. We can see from Fig.3 that the speed-up varies from 1 to 6 depending on the block size. The throughput increases as we use larger block sizes. But when the block size is 64K bytes, the performance is almost identical. This is because the hierarchy cache driver bypasses all IO request packets that are greater than 64K.

4.2.2 ZDESTT

ZDESTT is Ziff Davis email server test tool. It is based on the client/server architecture. The test platform includes the mail server, the controller and five clients. They are connected through 100Mbps Ethernet switch. During the tests, the clients send POP, SMTP or IMAP requests to the mail server and record the response time. All the results are reported to the controller that analyzes the results.

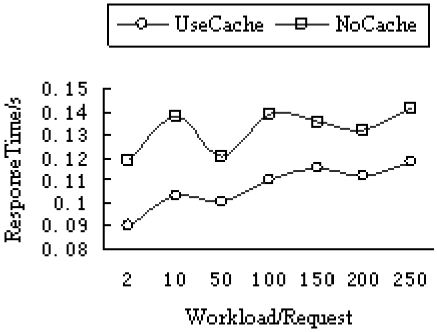


Fig. 4. Departmental POP Test Result

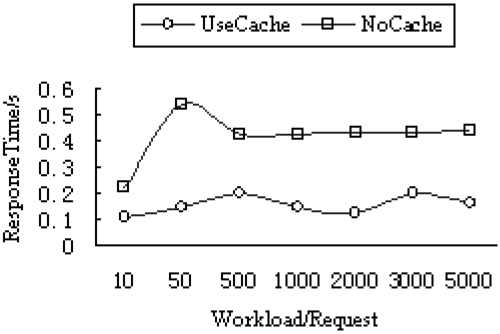


Fig. 5. Enterprise POP Test Result

We use the standard test suites: departmental POP and enterprise POP. The latter is a heavier workload. But the characteristic of both test suites is same. The test clients will spend 70% of time on logging on the email server, and retrieving the emails from randomly selected accounts using POP3 protocol. In this phase, all the requests are read operations. And the test clients will spend 30% of time on sending emails to the test accounts using SMTP protocol. In this phase, almost all requests are write requests. In these tests, the IO requests are bursty.

We can see from Fig. 4 and Fig. 5 that the hierarchical cache driver can improve the email server performance dramatically. On one hand, in the test using Departmental POP suite, the response time is reduced by approximately 30%. On the other hand, the prototype hierarchical cache driver achieves a performance improvement as high as a factor of 3.6 over the traditional device driver in the test using the enterprise POP suite. In the latter test suite, the workload is heavier, and the small writes is more intensive. The experimental results show that the more intensive the small write requests are, the more effective the hierarchical cache driver is.

4.2.3 IOMETER

IOMETER is an IO performance analysis tool that was first developed and maintained by the enterprise server group of Intel Corporation. IOMETER is both a workload generator and a performance measurement tool. IOMETER simulates workloads to stress the IO subsystem in specific ways. Under a simulated workload, IOMETER gathers data such as throughput, latency, and CPU utilization. By running the same workload on multiple system configurations, users can determine the optimum configuration. IOMETER can generate and measure loads on single or multiple (networked) systems.

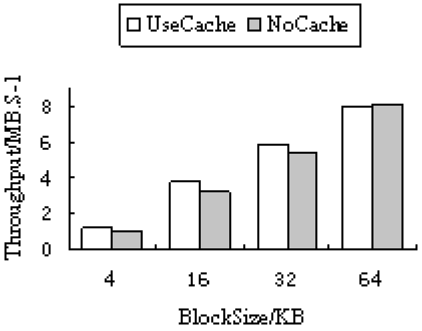


Fig. 6. IOMETER Test Results Comparison

In the tests, IOMETER worker of dynamo generates sustained IO requests to evaluate the sustained IO performance. The results reflect the extreme capability of the IO subsystem. The workload settings are 100% random, 100% write, and one hour of test duration. We performed tests with different block size on the two configurations: one with the hierarchical cache filter driver and the other without. Fig. 6 shows slight performance improvement for the hierarchy cache. It improves the sustained IO performance by about 15% when the block size is 4K. Compared with the previous tests, the performance gain is limited. This is because that IOMETER sends IO requests continuously in the tests so that the system keeps busy all the time. There is no idle time for the idle destaging but forced to destage. The performance improvement results mainly from the reduced rotation and seeks time. The performance is identical when the block size is 64K, because the filter driver will not buffer the IO request that is equal to or larger than 64K.

5 Conclusion

We have presented in this paper a design, implementation, and performance measurements of a new disk cache architecture. A pagefile and a small RAM cache are organized into a two-level hierarchical cache for disk accesses. Small writes are first collected in the RAM cache. When the system is idle, the data will be destaged from the RAM cache to the pagefile in large writes. The hierarchical cache exploits the performance difference between difference data sizes and different ways of accessing disks. One specific implementation has been carried out as a filter driver on the Windows 2000/Windows XP. Experimental results show that the hierarchical cache can improve the small write performance by a factor up to 6 for traffic intensive small write requests. It can increase the mail server performance dramatically when there are intensive random small writes. Moreover, the hierarchical cache driver is completely transparent to the file system and physical device. It does not require any modifications to the original OS nor the existing partition layout. It therefore can be inserted into the existing disk driver stack to obtain immediate performance gain.

Acknowledgements. The authors would like to thank the anonymous referees for their valuable comments on the original manuscript. Their detailed suggestions improve the quality of this paper greatly. This research is mainly sponsored by National Natural Science Foundation of China (No.60273073 and No. 60173043). Dr. Qing Yang's research is supported in part by National Science Foundation of USA under grants CCR-0073377 and CCR-0312613 and Elake Data Storage System Corporation.

References

1. Richard Stacpoole and Tariq Jamil, Cache memories – Bridging the Performance Gap, IEEE POTENTIALS, April/May 2000.
2. Y. Hu and Q. Yang, A New Hierarchical Disk Architecture, IEEE Micro, Vol. 18, No. 6, November/December 1998.
3. Y. Hu and Q. Yang, DCD-Disk Caching Disk: A New Approach for Boosting I/O Performance, 23rd Annual International Symposium on Computer Architecture, Philadelphia PA May, 1996, pp.169-178
4. P. Chen, E. Lee, G. Gibson, R. Katz, and D. Patterson, RAID: High-Performance, Reliable Secondary Storage, ACM Computing Surveys 26(2), 1994, pp.145-185
5. K. Treiber and J. Menon, Simulation Study of Cached RAID5 Designs, Proceedings of International Symposium on High Performance Computer Architectures, Jan. 1995, pp. 186-197
6. D. Stodolsky, M. Holland, W. V. Courtright II, and G. A. Gibson, Parity Logging Disk Arrays, ACM Transaction of Computer Systems, pp. 206-235, Aug. 1994.
7. K.H. Yueng and T.S. Yum, Dynamic Parity Logging Disk Arrays for Engineering Database, IEE Proc.-Comput. Digit. Tech., Vol. 144, No. 5, September 1997
8. J. Ousterhout and F. Douglas, Beating the I/O Bottleneck: A Case for Log-structured File Systems, Technical Report, Computer Science Division, Electrical Engineering and Computer Sciences, University of California at Berkeley, Oct. 1988.

9. Rajeev Nagar, Windows NT File System Internals: a Developer's Guide, ISBN: 1-56592-249-2, O'Reilly & Associates, 1997.
10. Daniel Robbins, Advanced File System Implementor's Guide, URL: http://www-106.ibm.com/ developerworks/linux/library/l-fs7, April 2001.
11. John L. Hennessy and David A. Patterson, Computer Architecture-A Quantitative Approach (Third Edition), ISBN: 1-55860-596-7, Elsevier Science Pte Ltd., 2003
12. T. Nightingale, Y. Hu and Q. Yang, The Design and Implementation of a DCD Device Driver for Unix, 1999 USENIX Technical Conference, Monterey, CA, June, 1999
13. Q. Yang and Y. Hu, Disk Caching Disk: A New Device for High Performance I/O System, U.S. Patent and Trademark Office, No. 5,754,888, Approved September 24th, 1997.

An Object-Oriented Data Storage System on Network-Attached Object Devices*

Youhui Zhang and Weimin Zheng

Institute of High Performance Computing Technology
Dept. of Computer Science, Tsinghua Univ.
100084, Beijing, P.R.C
zyh02@mail.tsinghua.edu.cn

Abstract. This paper presents a cluster-based storage platform--OStorage that employs Network-Attached Object Storage Device (NAOSD) as the low-level storage device. Owing to some features of NAOSD, including object-like access interface, simple computing abilities and self-management, OStorage supports structured-data directly to eliminate the data-model-mismatch problem of conventional storage systems. In addition, the OStorage prototype implements distributed data access, distributed transaction, some query functions and can simplify the building of scalable Internet services. The performance analysis shows that its access time increases with the system scale logarithmically, which is better than the conventional systems. And experiments show that its scalability is fairly satisfying.

1 Introduction

Today's Internet services demand the storage platform to posses many features including the ability to scale to large, high availability in the face of partial failure and operational manageability. It is challenging for a storage platform to achieve all of these properties. Many projects propose using software platforms on clusters to address these challenges.

Lore[1] is a database management system (DBMS) for XML from Stanford. The project focuses on defining a declarative query language for XML and developing new technology for interactive searches over XML data. Porcupine [2] provides one transactional record store, which combines the simplicity and manage -ability of the file system interface with a select few features for managing record-oriented data. Ninja [3] project implements DDS (distributed data structure) that presents a conventional single- site data structure interface to service authors, but partitions and replicates data across a cluster. Now a distributed hash table DDS is implemented.

However they use traditional block-based disks as low-level storage devices to construct new storage systems, which will cause a data-model-mismatch problem between applications and the storage systems. Because applications access storage through server bottlenecks[4], that a single "server" computer copies and converts data between the storage (peripheral) network and the client (local area) network.

* Supported by High Technology and Development Program of China (No. 2002AA1Z2103).

P.-C. Yew and J. Xue (Eds.): ACSAC 2004, LNCS 3189, pp. 527–538, 2004.

As processor performance increases and memory cost decreases, system intelligence continues to move away from the CPU and into peripherals. Storage system designers use this trend toward excess computing power to perform more complex processing and optimizations inside storage devices. Some research projects, including NASD[5], Attribute based Storage[6], Active Disks[7], bring forward the idea of Object-based Storage Device (OSD), which means that some overloads owned by traditional file servers are offloaded to peripheral storage devices.

Enlightened by this idea, we design a new network-attached object-based storage device (NAOSD). Compared with the previous projects, NAOSD supports structured-data storage directly to eliminate the data-model-mismatch problem. It can move portions of a server's processing to a storage device to improve the system scalability.

Then the usage of the prototype of NAOSD in cluster storage environments, OStorage, is proposed. It owns the following features,

- OStorage is an object-oriented data management layer as a cluster infrastructure software with transaction support, specifically for the construction of Internet services.
- Peer-Client and modular design are adapted. Many storage properties, including the storage capacity, network bandwidth and throughput, are highly scalable.
- OStorage supports transparent persistence in the Java programming language. The client interface is compatible with Java Data Objects API [8], an emerging standard for transparent data access in Java.
- Data & Meta-data uniform storage and query location mechanisms are introduced to improve the system scalability. The performance is analyzed in Part 3.4.

Now, OStorage has been achieved based on NAOSD prototypes. The usage shows that its scalability of throughput is nearly linear. The rest of this paper is organized as follows: Section 2 describes the features of NAOSD and system architecture of OStorage. The detailed design and implementation are introduced in section 3 that also presents the performance analyses. Section 4 gives the performance data and the last part summarizes this paper.

2 Cluster-Based Object Storage

2.1 Features of NAOSD

NAOSD owns the following basic features,

- Object-like access interface: Object data, the variable-length data unit and its different attributes, is supported by NASOD directly. Moreover, NAOSD has the ability to parse the object data to get one or more field values of its structure.
- Simple Computing abilities: Query and sorting are both supported by NAOSD, which act as a filter to data as it moves from the disk to upper-level services. This reduces the amount of data on the interconnect and offloads the host processor. The simplest example of this is a set select operation.
- Data and meta-data uniform storage: Object field values can also been used to define different attributes of the object, which contain the access mode, its priority of I/O and so on. That is, meta-data is a part of the object data.

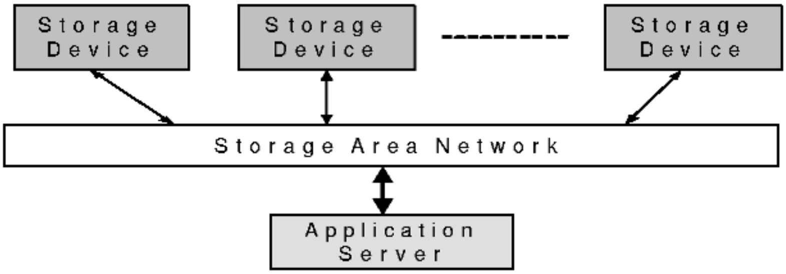


Fig. 1. IO-dense data access model

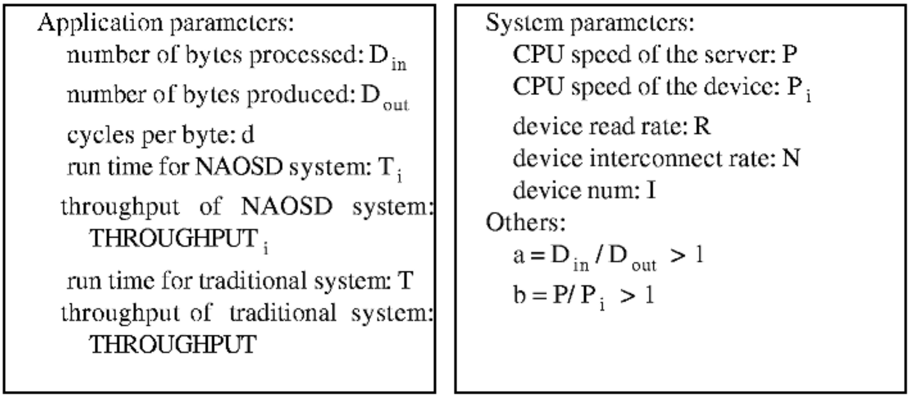
2.2 Potential Benefits of NAOSD

Three advantages are introduced when using NAOSD as the storage device:

- Direct storage-device-to-service transfers are supported to eliminate the traditional server bottleneck.
- Some server functions are ported to the storage devices to leverage the parallelism available in systems with large numbers of disks.
- The amount of data on the interconnection is reduced. It is specially useful for those data-intensive Internet services.

In the following IO-dense data access model, the performance of a server system with a number of "dumb" block-level devices is compared with the same system with the traditional devices replaced by NAOSD to determine the potential benefits.

As showed in Figure 1 the IO-dense service is running on the server. The service receives requests, processes the data from storage devices and returns the last result to requesters. Some parameters in Figure 2 are introduced to describe the whole model. We assume that d is so small that data can been processed distributedly by devices.



Application parameters:
number of bytes processed: D_{in}
number of bytes produced: D_{out}
cycles per byte: d
run time for NAOSD system: T_i
throughput of NAOSD system: $THROUGHPUT_i$
run time for traditional system: T
throughput of traditional system: THROUGHPUT

System parameters:
CPU speed of the server: P
CPU speed of the device: P_i
device read rate: R
device interconnect rate: N
device num: I
Others:
$a = D_{in} / D_{out} > 1$
$b = P / P_i > 1$

Fig. 2. Parameters of distributed data access model for IO-dense applications

T_i and T are illustrated as follows.

$$T_i = \max\left(\frac{D_{in} / I}{R} , \frac{D_{out}}{N} , \frac{(D_{in} / I) * d}{P_i} \right)$$

$$T = \max\left(\frac{D_{in} / I}{R} , \frac{D_{in}}{N} , \frac{D_{in} * d}{P} \right)$$

That is, run time is determined by the minimum among device read rate, device interconnection rate and the CPU speed of the device if the data process is parallelized ideally.

Then the system throughput can been computed.

$$\mathrm{THROUGHPUT}_i = \frac{D_{in}}{T_i} = \min\left(R*I, a*N, \frac{P * I}{d * b} \right)$$

$$\mathrm{THROUGHPUT} = \frac{D_{in}}{T} = \min\left(R*I, N, \frac{P}{d} \right)$$

Therefore, some conclusions can been drew based on different assumptions.

1) Device read rate (R*I) is the minimum.

$$\frac{\mathrm{THROUGHPUT}_i}{\mathrm{THROUGHPUT}} = I / b$$

It means that the performance benefit will be introduced if the amount of processing capacities of all NASODs exceeds that of the server.

2) CPU speed of the device (P/d) is the minimum.

$$\frac{\mathrm{THROUGHPUT}_i}{\mathrm{THROUGHPUT}} = I / b$$

The conclusion is as same as 1).

3) Data transfer rate (N) is the minimum.

$$\frac{\mathrm{THROUGHPUT}_i}{\mathrm{THROUGHPUT}} = \min\left(\frac{R * I}{N} , a, \frac{P * I}{d * b * N} \right)$$

We know,

$$\frac{R * I}{N} > 1,\ a > 1 \text{ and } \frac{P * I}{d * b * N} > I/b.$$

So, it is still determined by the comparison between the amount of processing capacities of all NASODs and that of the server. Now many large scale storage systems, including Compaq TPC-C, Microsoft Terra- Server, Digital TPC-C and Digital TPC-D 300, satisfy this condition.

2.3 Overview of OStorage

OStorage system is defined as a self-contained object-oriented data management layer running on a server cluster to handle storage requests of Internet services running on the same cluster. The clients in Figure 3 connect to service instances through Internet by Internet application protocols such as HTTP, IMAP, etc.

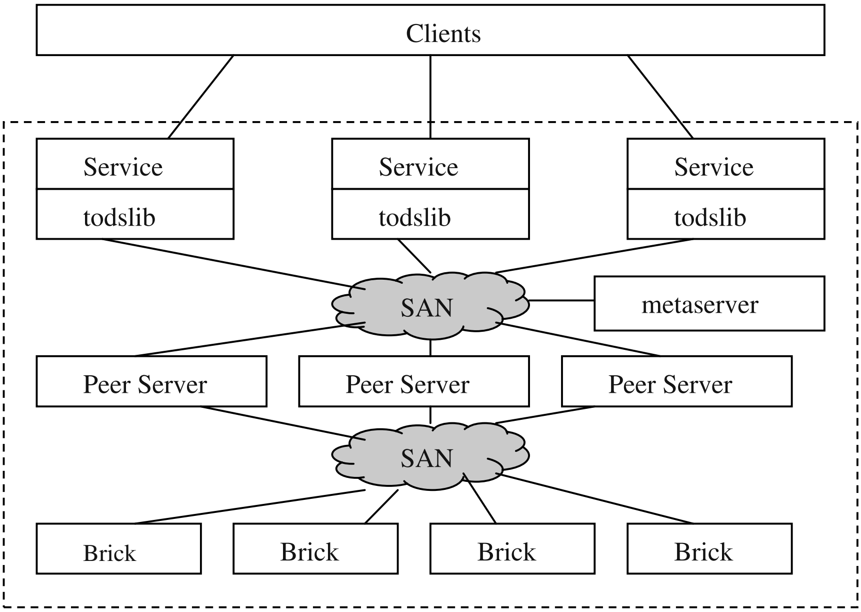


Fig. 3. OStorage overview

Services are multiple instances of the same Internet service. They connect to OStorage components known as Peer Servers to access data. They are inherently identical to each other from users' view, each presenting a single image of the whole system and they communicate with each other in a peer-to-peer style.

There are also other components (the lower blocks, named Brick), which the services do not connect to directly. Brick is the instance of NAOSD that provides storage and query interfaces for structured-data employing the power of embedded processors.

Within the service process, a library named TODSLib maps user API calls to messages sent to Peer Servers and parse results from them. Currently a Java version of TODSLib is implemented. As mentioned before, it implements transparent persistence and is compliant with the Java Data Objects API.

The Meta-Server maintains system configuration and meta-data. It is replicated and thus assumed fault-tolerant, providing a safe place for critical global information. System configuration includes location (IP, port) and parameters of all components such as Peer Servers and Bricks. This ensures centralized management of the whole system.

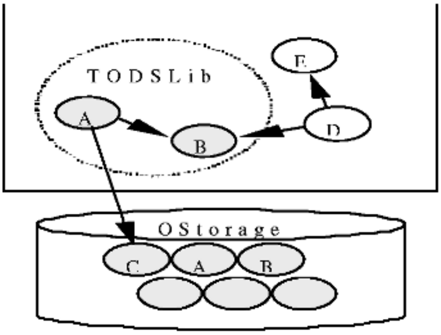


Fig. 4. Object references

3 Design and Implementation

3.1 Data Model

Objects managed by OStorage are put into name spaces known as Object Spaces. Each object space has its own set of class hierarchy and objects. An Object Space is analogous to a database or a table space in RDBMS, or a directory in file systems. The list of all Object Spaces, class meta-data and permission rules of each Object Space are all maintained by the Meta-Server. Data of an Object Space are stored on a subset of all the Bricks, whose list is managed by the Meta-Server. Object (or persistent object) is the granularity of most operations in OStorage. Every persistent object is associated with a OID. An object has a number of value fields and can reference other objects. Figure 4 illustrates an example of object references. A and B are active persistent objects, with A referring to an inactive one, C. D and E are transient objects not currently managed by TODSLib. However, they will become persistent by a make persistent call to TODSLib.

When a transient object is made persistent, all objects reachable from it are also made persistent (persistence by reachability). Persistent objects are long-lived and independent of life-cycles of the service instances or TODS runtime. Any modification to the object will be written to the store implicitly at some time (e.g., when transaction is committed). Persistent objects are loaded into memory automatically when needed.

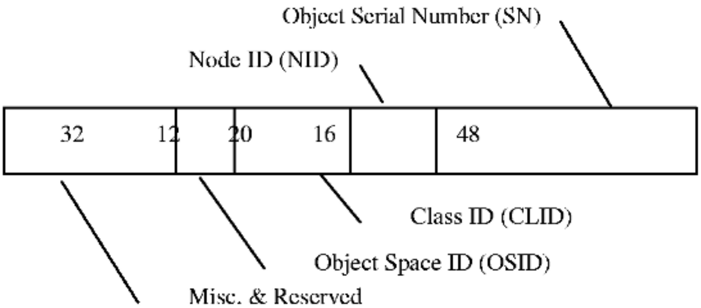


Fig. 5. 128-bit OID format

3.2 Object Identification

An Object ID is a 128bit integer, whose structure is shown in figure 5. Object Space ID (OSID) indicates which Object Space this object belongs to. Class ID (CLID) references to class definition in the Meta-Server, it is assigned when first object of its class is inserted into the system. Node ID (NID) denotes the location of the object, while Serial Number is the local ID of the object.

One thing to notice is that the OID is a physical ID, in the sense that it indicates on which node the object is located. The system can directly find the object just by the OID. This contrasts to the alternative approach of using a logical object ID or "path" and thus needs to look up the real location of objects before accessing them, which introduces more overhead and the problem of effectively and coherently caching the lookups. Logical ID or text path are often introduced for user friendliness and location transparency. The former is not a problem in OStorage because TODSLib completely hides from users the details of fetching and storing persistence objects. OIDs are not even seen by them. The latter reason is most justified for wide-area distributed systems, where nodes and network failures and changes are common. As OStorage is designed for well managed cluster environment, it is found that an OID with more information greatly simplifies system design and improves performance.

3.3 Access Interface

Moreover OStorage supports non-transactional and transactional modes of access. It is determined by whether data accesses are enclosed in a transaction.

For non-transactional accesses, data caching on Peer Servers are enabled, which results in much better performance. However there is no guarantee about data consistency under concurrent access. Different Peer Servers may report different value for the same object at some moment due to asynchronous cache invalidation. Consistency level of non-transactional access is PRAM Consistency [9], using parallel computing terminology, i.e., writes made by each specific client are seen by others in the original order, but the global order is not guaranteed.

Both Peer Servers and Bricks are designed to be transactional. Distributed transactions on multiple Bricks are managed using the two-phase commit protocol. In current prototype NAOSD is simulated on Berkeley DB[10], which is a embedded database with commit/abort functions. When distributed transaction must be done, the corresponding Peer Server acts as the transaction manager, and participating Bricks act as resource managers. All status information of ongoing distributed transactions is stored persistently in a simple database managed by the Peer Server, in order to make both Peer Server and Brick failures recoverable.

Bricks provide upper-level services with the following interfaces:

```
BeginTransaction, prepareTransaction, commitTransaction,
rollbackTransaction
```

In addition to the location mechanism through OID, distributed query is achieved to fetch objects from Bricks. It looks like a cursor operation in DBMS, which is managed by one Peer Server to command Bricks to filter objects according to some field condition. Then, services can browse all objects returned.

One important feature of OStorage is that some common functions of services are implemented in the storage layer, and NAOSD plays an important hole to simplify this achievement.

3.4 Performance Model

Data & meta-data uniform storage and query location mechanisms are introduced in previous sections and we give a performance model for them to argue OStorage owns higher scalability as compared with conventional systems.

Our model contains N Peer Servers and n Bricks connected with a high performance network. The CPU speed ratio of Peer Server and Brick is m (m>1) and a fixed amount of objects is stored in every Brick. In addition, the meta-data used to locate objects can be stored in two ways. First, they are stored as the fields of data on Bricks in uniform storage mode. Second, they are separated from the data and placed distributedly on every Peer Server in the conventional mode

Then, performance models of these two modes are computed respectively. At first the data location flow of the conventional mode is described in Fig 6 and the flow of uniform storage in Fig 7.

Some parameters are introduced to describe models.

T_{lookup} : time for Brick to browse the local meta-data, which is in direct ratio to the number of objects and in inverse ratio to the CPU speed. Then time used by Peer Server to query the local meta-data is $T_{lookup} * \frac{n}{N*m}$.

T_{access} : time used to access the local data and meta-data.

T_{remote} : the point-to-point communication speed between nodes.

$T_{p-broadcast}$: the broadcast speed among Peer Servers.

$T_{b-broadcast}$: the broadcast speed between one Peer Server and all Bricks.

Time to locate one object in the conventional mode, $T_{traditional}$, can be presented as the following equation.

$$T_{traditional} = T_{lookup} * \frac{n}{N*m} + T_{access} * \frac{1}{N} + (T_{p-broadcast} + T_{lookup} * \frac{n}{N*m} + T_{remote} + T_{remote})* \frac{N-1}{N} + (T_{remote} + T_{access})$$

The first term is the time spent by the Peer Server on looking for the local meta-data. Its hit probability is 1/N, so the next term is the time to access the hit meta-data. The third is the overhead of transmitting the location request to other Peer Servers if the meta-data is missed locally. The last is the time spent by the target Brick on reception and accessing the located object.

$$T_{traditional} = (\frac{2nN-n}{m*N^2})* T_{lookup} + 2* T_{access} + \frac{N-1}{N} * T_{p-broadcast} + \frac{2N-1}{N} * T_{remote}$$

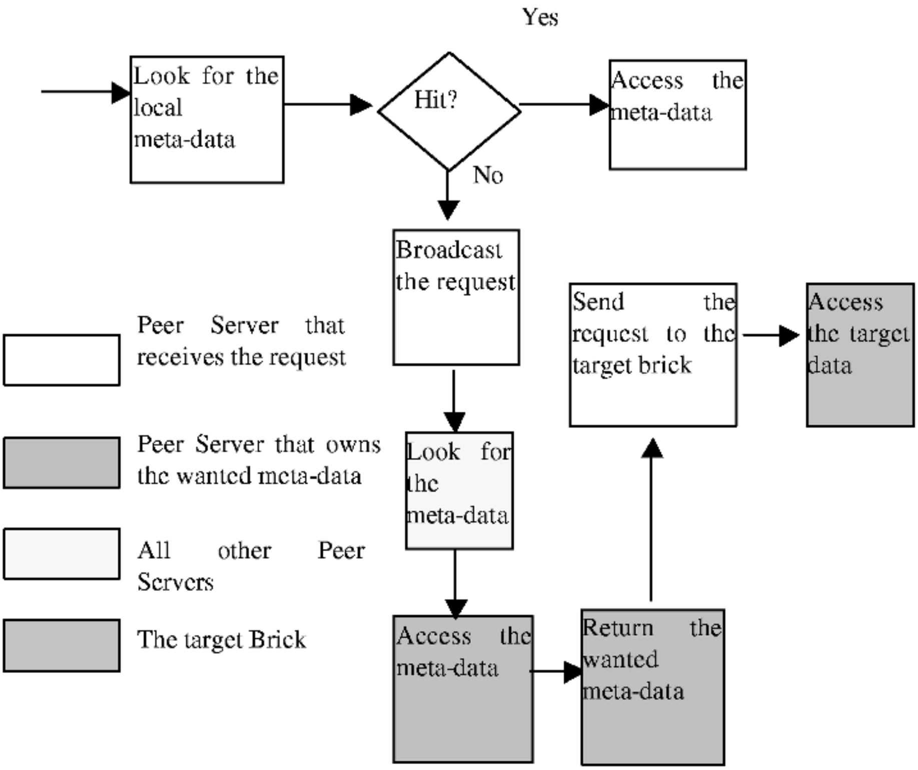

Look for the local meta-data
Hit?
Yes
No
Access the meta-data
Broadcast the request
Send the request to the target brick
Access the target data
Look for the meta-data
Access the meta-data
Return the wanted meta-data
Peer Server that receives the request
Peer Server that owns the wanted meta-data
All other Peer Servers
The target Brick

Fig. 6. The data location flow of the conventional mode

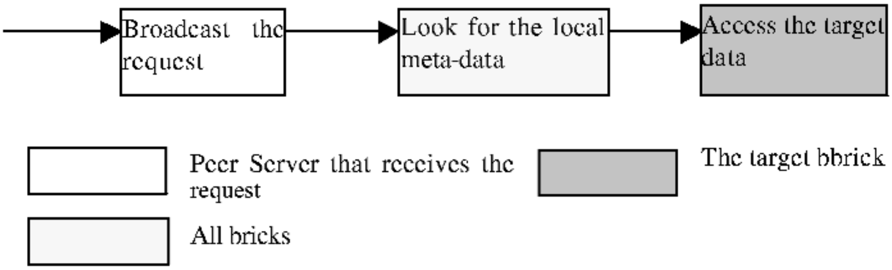


Fig. 7. The data location flow of uniform storage and query location mechanisms

In the uniform storage mode time to locate one object, T_{naosd}, is computed by the next equation.

$$T_{naosd} = T_{b-broadcast} + T_{lookup} + T_{access}$$

The first term is the overhead for one Brick to broadcast the location request to all others. The next two correspond to time consumed by every Brick to look for and access the local data respectively.

To simplify the analysis, T_{remote} is assumed a constant and binomial tree algorithm [11] is adopted for the broadcast communication. So broadcast overhead can be described as following equations.

$$T_{p-broadcast} = (\lfloor \mathrm{Log}_2(N-1) \rfloor + 1) * T_{remote}$$
$$T_{b-broadcast} = (\lfloor \mathrm{Log}_2(n-1) \rfloor + 1) * T_{remote}$$

Some other assumptions are also introduced as follows.

$$T_{remote} = 5,\ T_{lookup} = 1,\ T_{access} = 10,\ N = 5,\ m = 5.$$

Based on these hypotheses, $T_{traditional}$ and T_{naosd} can be computed.

$$T_{traditional} = \frac{9n}{125} + 20 + 0.8 * 5 * 3 + 9 = \frac{9n}{125} + 41$$

$$T_{naosd} = (\lfloor \mathrm{Log}_2(n-1) \rfloor * 5) + 16$$

So, $T_{traditional}$ increases with the system scale linearly while T_{naosd} is logarithmical.

3.5 Prototype and Its Usage

The prototype of OStorage is implemented and its hardware platform contains a cluster of PC servers connected with 100M Ethernet. The whole system is coded with JDK 1.4 other than Bricks that are implemented in C language based on Berkeley DB.

The following features are achieved.

- Structured-data storage functions, including object read/write/create/delete.
- Soft consistency replication and linearizability are both implemented.
- Distributed transaction and query.
- Meta server is employed to manage the whole system to provide a global consistent storage view of Bricks for Peer Servers.
- Requests between TODSLib and Peer Servers can been sent in a burst mode, that is, one session is connected at first and several requests can been transferred in one time if possible to decrease the overload.

4 Results

Performance experiment results are presented in this section. Our test environment is a 36-node server cluster and each node is equipped with 4 Intel Pentium III Xeon processors at 750 MHz, 1 GB of RAM and a 36 GB 10000 RPM SCSI disk. The network is 100M fast Ethernet. All nodes run Redhat Linux 7.2. The system and test programs were run with Sun JDK 1.4.0-b92 for x86 Linux.

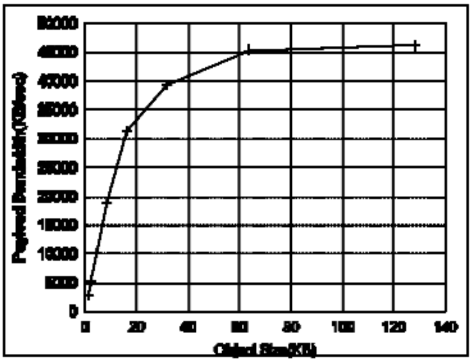


Fig. 8. On-disk Read Payload

4.1 On-Disk Reads

This test is closer to actual operational environment. To approximate real-world workload, we first populated each of the Bricks with 5000 objects of the object length being tested. Then we access these objects randomly by Object ID we gather when inserting them. Although random access is not a good "real-world" pattern, it effectively shows the bottom-line performance we should expect. The Object Caches in Peer Servers are turned off to show raw Brick read performance. It completes as many as 2740 reads of 1KB object in a second. As object size increases, payload bandwidth increases quickly, to 46MB/s when object size is 128KB. More detailed results are showed in Figure 8. This throughput result is satisfying. Since actual work-load usually has good locality, the efficiency of the Buffer Cache will be much better, thus overall throughput higher.

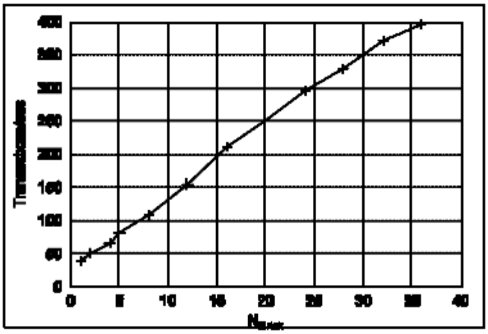


Fig. 9. Transactional Write

4.2 Transactional Writes

Transaction performance is directly tied to disk write performance because they include synchronous writes to the log file. Here we test inserting objects into Bricks by transactions. In each transaction, we insert four objects that are about 2K in size totally. From the results shown in Figure 9, we can see that the transaction performance grows linearly with Brick number, just as we expected. When all 36 Bricks participate in, 396 transactions can be done in a second.

5 Conclusion

In this paper, we present the design and implementation of OStorage, a cluster storage platform for Internet services. It is designed with the requirements of scalable services in mind and appeals to many advantageous properties of modern server clusters. One type of OSD, NAOSD is used as the low-level storage device, which supports structured-data directly to eliminates the data-model-mismatch problem. In addition, moving portions of an service's processing to a storage device significantly reduces data traffic and leverages the parallelism already present in large systems.

The user interface implements transparent persistence which relieves developers completely from writing I/O code. Different levels of availability are supported that make OStorage suits the requirements of different services.

References

1. R. Goldman, J. McHugh, and J. Widom. From Semistructured Data to XML: Migrating the Lore Data Model and Query Language. Proceedings of the 2nd International Workshop on the Web and Databases, Philadelphia, Pennsylvania (1999).
2. Robert Grimm, Michael M. Swift, and Henry M. Levy. Revisiting structured storage: A transactional record store. Technical Report UW-CSE-00-04-01, University of Washington, Department of Computer Science and Engineering (2000).
3. Steven D. Gribble, Eric A. Brewer, Joseph M. Hellerstein, and David Culler. Scalable, Distributed Data Structures for Internet Service Construction. In Proceedings of OSDI 2000. San Diego, CA (2000).
4. Garth A. Gibson, David F. Nagle, William Courtright II, etc. NASD Scalable Storage Systems. In the Proceedings of USENIX 1999, Linux Workshop, Monterey, CA (1999).
5. G. A. Gibson, et al., A Case for Network Attached Secure Disks, Tech. Report CMU-CS-96-142, Carnegie Mellon University (1996).
6. Elizabeth Shriver, A Formalization of the Attribute Mapping Problem. HP Labs Technical Reports, HPL-1999-127.
7. Erik Riedel, Christos Faloutsos, Garth A. Gibson, etc. Active Disks for Large- Scale Data Processing. IEEE Computer, Vol.34, No.6 (2001). 68-74.
8. C. Russell, Java Data Objects 1.0 Proposed Final Draft, JSR12, Sun Microsystems Inc., available from http://access1.sun.com/jdo (2001).
9. M. Raynal and A. Shiper. A Suite of Formal Definitions for Consistency Criteria in Distributed Shared Memories. ISCA Proceedings of the International Conference PDCS, Dijon France (1996). 125-130.
10. Sleepycat Software Inc., Berkeley DB Programmer's Tutorial and Reference Guide, available at www.sleepycat.com (2001).
11. Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. Introduction to Algorithms[M]. MIT Press (1990).

A Scalable and Adaptive Directory Scheme for Hardware Distributed Shared Memory

Kiyofumi Tanaka[1,2] and Toshihide Hagiwara[1]

[1] School of Information Science, Japan Advanced Institute of Science and Technology,
1–1 Asahidai, Tatsunokuchi, Ishikawa, 923–1292 Japan
[2] “Information and Systems”, PRESTO, Japan Science and Technology Agency
{kiyofumi,t-hagiwa}@jaist.ac.jp

Abstract. In hardware distributed shared memory in the style of CC-NUMA, directory information that specifies locations of sharing processors is used for cache coherence. Structure of the directories affects the size of hardware, time required for coherence transaction, and network traffic. In this paper, we propose and evaluate a new scalable directory scheme, ”*adaptive hierarchical coarse directory*”, that exploits hierarchy in the system and exhibits appropriate values in terms of the above three items. The directory has tolerance to many copies of a memory block scattered in a large-scale parallel system. This characteristic makes it easy for operating systems to allocate parallel threads in multitasking/multiuser environment.

1 Introduction

In a distributed shared memory (DSM) system based on CC-NUMA (Cache Coherent Non-Uniform Memory Access) system, sharing information must be managed per block such as cache line, page, or object. Sharing information indicates whether or not the block is shared and whether it has been updated or not, and includes a directory that identifies the processors holding a copy of the block. A directory scheme in hardware DSM affects the amount of hardware, efficiency of coherence processing, and network traffic. For example, when the amount of memory required for storing directories increases in proportion to the number of processors in a system, the amount might get larger than that of general-purpose program data, which is unrealistic from the point of view of efficient use of memory. Therefore, it is difficult to apply the kind of directory to a large-scale system.

We proposed a small-size directory scheme and communication methods cooperating with the directory, and built a prototype parallel computer which implemented them [1]. The preliminary evaluation based on the information gained from the prototype showed that the directory with the communication methods had better scalability on the amount of memory required, efficiency of coherence processing, and network traffic in a large-scale system than other existing directories such as a full-map directory scheme [2]. However, the directory has a possibility of generating more traffic than other directory schemes when the

P.-C. Yew and J. Xue (Eds.): ACSAC 2004, LNCS 3189, pp. 539–553, 2004.

number of processors sharing a block is not large and the copies of the block are scattered in the system. In this paper, we propose a new directory scheme which is adaptive to the situation where our previous scheme might increase redundant communications, and evaluate it quantitatively by using real programs.

In section 2, scalability issues of directory schemes in hardware DSM are discussed. Section 3 describes the outline of a lightweight hardware DSM we proposed. In section 4, we propose a new directory scheme. Section 5 compares our scheme with other schemes in terms of the size, time, and traffic required for coherence processing. Section 6 describes the related work and Section 7 concludes this paper.

2 Directory Schemes of Hardware DSM

The size of memory taken up by directories increases as the scale of the system is enlarged. Therefore, the structure and size of a directory will affect the hardware costs. Directory schemes are classified into two types: one type completely identifies the processors that hold a copy of a memory block, and the other incompletely. The former has a problem in that it requires a large amount of memory for directory storage when there are many processors in a system, or a problem in that an overhead of accessing a directory is large when the directory size is larger than the bit width of a memory component or when the structure of the directory is based on a linked list. On the other hand, there are two general schemes for the latter: one in which the number of processors that can share a block is limited and the other in which the processors that share a block are identified roughly, that is, a group that includes all the processors which share a block is indicated. Both take up relatively less memory than complete identification methods but there are still significant overheads caused by broadcasting or multicasting of coherence messages when many processors share a block.

Full-map directory [3], LimitLESS directory [4], chained directory [5,6] and hierarchical bit-map directory [7] hold complete information on sharing. The full-map directory assigns one bit to each processing node to indicate whether the processing node holds a copy of the relevant memory block. Since this scheme requires memory in proportion to the number of processing nodes, it is not suitable for large-scale systems. Multistage accesses to the directory memory are also required for getting one directory's information if the number of processing nodes exceeds the width of a single access, for example, 64 or 128 bits.

The LimitLESS directory places limitation on the number of processors that can share a block in order to reduce the memory requirement. The directory has the limited number of pointers to point to processors with the block copy. When the number of copies exceeds the limit, a protocol processor or processing element emulates the full-map scheme. Although this directory requires less memory than full-map when the system has many processors, execution of the software brings a large overhead.

Although the size of a chained directory is small because it is a pointer, the structure generates long access latencies caused by sequential accesses to the linked directories. Therefore, it is important to keep the number of sharing processors low by employing an invalidation protocol when this directory is used.

In the hierarchical bit-map directory, sharing information for a memory block is distributed, that is, full-map information is partitioned into sub-bitmaps among hierarchical levels in a tree network. The directory size, therefore, increases with the scale of the system. It requires about $\sum_{k=0}^{m-1}(n+1)n^k$ bits for each shared memory block in an n-ary tree with height m. More memory is thus required than for a full-map directory. The scheme inherently increases communication latency, since it requires access to a part of the directory at every level of the hierarchy. Accordingly, the directory should be stored in high speed memory to prevent the high network latency from degrading system performance.

Consequently, when directories which have complete information on the locations of block copies are used in a large-scale system, the problems are that the directory is large, that access latency is high, or that protocol processing by a protocol processor/processing element induces large overheads of software execution.

On the other hand, the limited directory [8] and pseudo-full-map directory [9,10] obtain a size that is not proportional to the number of processors by using incomplete information on sharing. The limited directory uses a limited number of pointers to processors with a cached copy, and the directory size, therefore, does not increase in proportion to the system scale. When the number of copies reaches the limit, the next generation of a copy forces cache replacement by victimizing an existing copy or broadcasting of a coherence message to all processors, which leads to a lot of extra communication[1]. It is thus inevitable to keep the sharing number low by using an invalidation protocol.

The pseudo-full-map directory reduces the required memory by holding a bitmap per each level of the hierarchy in a tree interconnection network. There are three schemes in the pseudo-full-map directory, LPRA (Local Precise Remote Approximate), SM (Single Map), and LARP (Local Approximate Remote Precise). LPRA scheme specifies near processors as precisely as possible and more distant processors roughly. LARP specifies distant processors as precisely as possible, and nearer processors roughly. In SM scheme, all network nodes at a level use a unique bitmap. Here, directories for memory blocks are accessed only at the respective home processors because each directory can be maintained at its home processor. This directory scheme takes up the amount of memory proportional to $\log n$ when the system has n processors, and the incompleteness of sharing information leads to redundant communications in cache coherence transactions.

3 Lightweight Hardware DSM

We proposed a directory scheme, *hierarchical coarse directory*, which is of incomplete sharing information [1]. The size is smaller than any other existing

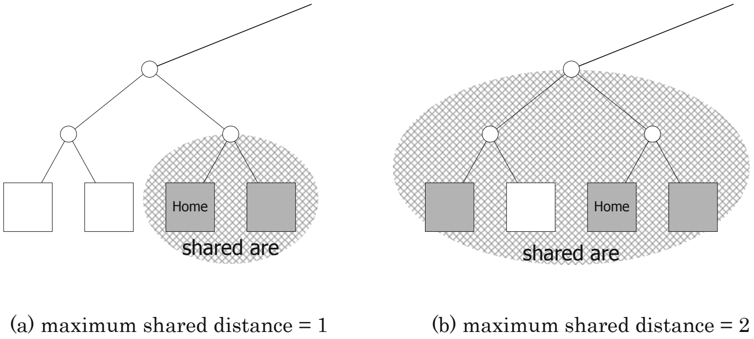
[1] Although the limited directory with broadcasting is of incomplete information, that with replacement policy is regarded as of complete.

directory schemes except ones based on broadcasting. Dynamic combining and multicasting mechanisms of an interconnection network cuts down messages increased by the incompleteness of the directory.

3.1 Hierarchical Coarse Directory

We assume that a tree structure can be physically embedded in the interconnection network. A home processor is statically assigned to each memory block. A home processor records a “maximum shared distance” as directory information. Here, the “maximum shared distance” is half of the number of hops between the home processor and the most distant processor with a copy of the block. In other words, the distance is the height of the minimum subtree which includes all processors that have a copy of the block.

Figure 1 is an illustration of the hierarchical coarse directory structure. The gray leaves in the figure represent processors which have a copy of the block[2], and the mesh areas indicate the shared area which includes all processors with a copy. In figure 1(a), the home processor and one of its next-door neighbors in the network hierarchy hold a copy. Therefore, the maximum shared distance is one. On the other hand, the maximum shared distance in figure 1(b) is two since the shared area is the subtree the height of which is two[3].



(a) maximum shared distance = 1 (b) maximum shared distance = 2

Fig. 1. Hierarchical coarse directory.

The shared area may include processors that don’t have a copy (for example, in the figure, the white leaves in the shared area). To the home processor, these processors also seem to have a copy. The simplicity of the directory representation causes an inaccuracy, that is, an overestimation of the number of block

[2] To be exact, a home processor has an original block. Here, we don’t distinguish between the original block and its copy.

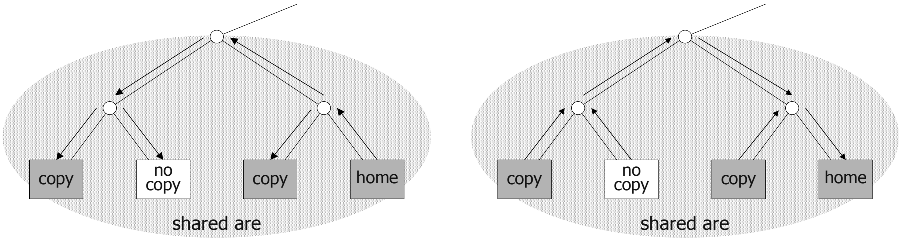
[3] In the actual hardware implementation, the maximum shared distance is smaller by one than the height of the subtree tree in order to simplify the calculating hardware. Therefore, the distance in figure 1(a) is zero, and that in figure 1(b) is one.

holders. However, the overestimation does not influence cache coherence. When a processor that does not have a copy receives a coherence message from the home processor, it has only to return a dummy acknowledgment message. The procedure maintains coherence.

The hierarchical coarse directory is $\lceil \log_2 \log_k n \rceil$ wide where n is the number of processing elements and the network has a k-ary tree structure[4]. For example, a four-bit directory for each memory block is sufficient to cover directory information for a massively parallel system that contains more than 64,000 processors connected by a binary tree network. Moreover, this directory system accomplishes the reduction of required memory without limitations on the number of copies.

3.2 Hierarchical Multicasting and Combining

Identical messages must frequently be transported to many or all processing elements during coherence processing such as invalidation or update. The transport performance can be improved by utilizing hierarchical multicasting. For example, when an invalidation is processed for a shared memory block, the home processor assigned in advance to the memory block issues only a single invalidation message. Each switching node in the network that has received the message multicasts it in all the directions that lead to any node within the shared area. Figure 2 (a) shows the multicasting operation when the maximum shared distance is two.



(a) multicasting

(b) combining

Fig. 2. Hierarchical multicasting and combining.

The directory scheme affects the multicasting method. In implementation of multicasting, a directory with complete information, such as a full-map directory, requires that a network packet for multicasting includes a group of destination processor numbers or the directory itself. The former approach makes the multicasting method impractical when the number of destination processors is large. For example, when the system has 1,024 processors that require 10 bits of a

[4] This is because the height of the tree can be decreased by one from the point of view of the actual hardware implementation, as mentioned before.

processor number to identify any processor, and 200 processors share a memory block, total 2,000 bits (250 bytes) must be included in the packet header, which is not small. The latter always requires the same number of bits as of all processors, which may be far from insignificant, and makes a switch perform an elaborate routing. On the other hand, the hierarchical coarse directory system is well-suited to multicasting. It is only necessary for network packets to include the directory information, that is, the maximum shared distance. The network switching nodes multicast by comparing the maximum shared distance with their own hierarchical level.

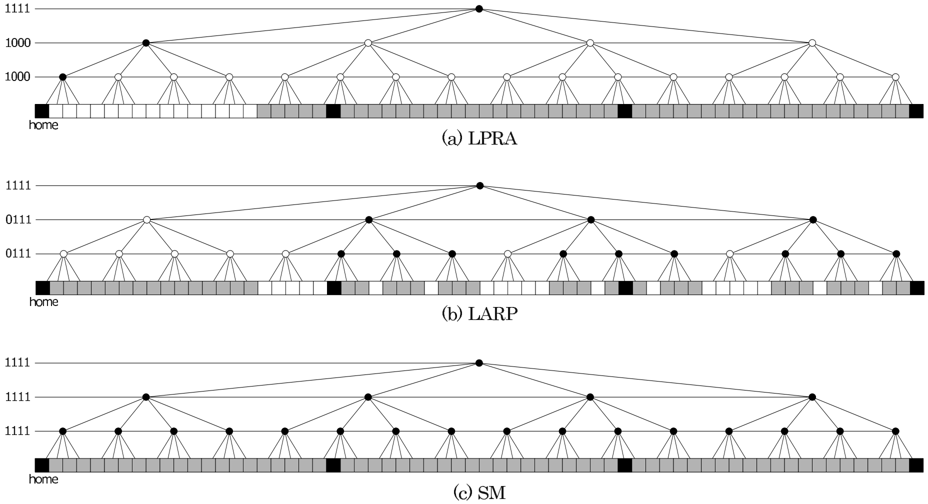
When two or more messages for an identical purpose are sent to a single processor, the messages can be combined into a single message at any level of the network hierarchy. This reduces the need for a series of processes at the destination processor. For example, every processor that has received an invalidation message returns an acknowledgment message which indicates the completion of invalidation processing within the processing node, even if it does not have a copy of the indicated block. All the acknowledgment messages are directed to the block's home processor. Each switching node forwards an acknowledgment message after it confirms the arrival of all acknowledgment messages from all branches along which it sent the invalidation message. The process is shown in Figure 2 (b).

The hierarchical coarse directory is also suited to the combining of messages. The number of messages to be combined at a switching node depends on whether the node is or is not a root of the shared subtree. When it is the root, the number is $k - 1$ where the network is k-ary. When it is not, the number is k. On the other hand, a switch must record the number of messages to be combined in some way for a full-map directory system.

The hierarchical multicasting and combining schemes require no serialized processing at a home processor. It takes only one round-trip latency between the home processor and the most distant processor in the shared area for the home processor to complete a coherence transaction, regardless of the number of processors with a copy. The combination of the reduced size of the hierarchical coarse directory and the hierarchical multicasting and combining make it possible actually to employ not only an invalidation protocol but also an update protocol even when there are a large number of sharing processors.

4 Adaptive Hierarchical Coarse Directory

The previous section indicated that the hierarchical coarse directory with multicasting and combining have quality of good scalability from the point of view of directory size and latency for a coherence transaction. However, more packets might be generated than other directory systems when copies are located irregularly or sparsely over the system, since the number of packets depends only on the hierarchical distance between a home and a most distant sharing processor, regardless of the number of sharing processors. This nature might influence other local processing in the system. In this section, we extend the hierarchical coarse



(a) LPRA

(b) LARP

(c) SM

Fig. 3. Dummy sharing processors in the pseudo-full-map directory.

directory and propose a directory scheme which brings light traffic even if copies are scattered.

The hierarchical coarse directory makes use of hierarchy of a tree interconnection network as the pseudo-full-map directory does. The two directories bring almost the same processing time for a coherence transaction by using multicasting and combining. However, directory size of the hierarchical coarse directory is smaller than that of the pseudo-full-map. This is because sharing information of the former is more incomplete than that of the latter. On the number of packets generated, the pseudo-full-map directory gives better results since it has a bitmap corresponding to each hierarchy. However, accurate (or somewhat accurate) information from the bitmaps can be applied only to either a path from a home to the root of the shared area or the other paths. Therefore, the number of packets does not depend on the number of copies.

Traffic generated by both of the directories is influenced by the locations of copies. When the number of copies is two in the hierarchical coarse directory or when that is n in the pseudo-full-map with n-ary tree network, there is a possibility that traffic occurs all over the network. For example, Figure 3 illustrates sharing processors when there are three copies in the pseudo-full-map directory system. The black leaves represent a home and processors which have a copy. The gray leaves are dummy sharing processors which don't have a copy but receive a coherence request. A bitmap for each hierarchy is applied to the black internal nodes at the corresponding level. In all schemes of the pseudo-full-map, that is LPRA (Figure 3(a)), LARP (Figure 3(b)), and SM (Figure 3(c)), three copies cause a number of network communications. The locations of copies depend on the distribution of data in a running program, the algorithm of the computation, or scheduling by an operating system. Although applying optimization for spatial locality to the program and the scheduling is effective in the locations, a directory system in which traffic is affected by the number of copies, not by the location of copies, is desirable when the optimization is difficult to apply.

Agarwal et al. [8] indicated that, in an invalidation protocol, the number of copies which should be invalidated when a block is updated is lower than four in most cases. This knowledge is used in the limited directory which limits the number of copies. However, when the limited directory is used simultaneously with an update protocol, the number of copies tends to exceed the limit and broadcasting frequently occurs, which leads to heavy traffic. A directory scheme we propose in this section has a limited number of pointers. When the number of copies is not more than the limited number, the pointers indicate processors which have a copy directly. When the number exceeds the limit, as many shared areas as the limit can be formed. When a processor which does not belong to any shared area joins the sharing group, one of the shared areas is extended so that the processor is included in the share area.

Let N be a number which corresponds to the limit of the limited directory. A directory consists of N pointer fields and $N + 1$ maximum shared distances. It does not include a pointer field for indicating the home because the location is implicitly known. Therefore, the number of pointers is smaller than that of maximum shared distances by one. Each pointer plays a role as a “home agent”, in other words, “base”. A home agent and a maximum shared distance related to the home agent form a “partial shared area”.

Initially, all pointers in a directory are invalidated. When a processor P starts to share a block and creates a copy of it, the home of the block reconstructs the directory according to the following algorithm.

```
if  (One of partial shared areas includes $P$.)
  - Perform nothing.
else if  (There is an invalid pointer field.) {
  - Set $P$ to the pointer field.
  - Set the maximum shared distance zero.
}
else {
  - Calculate hierarchical distances between any two among home, all home
    agents and $P$.
  - Select a pair (or a set of more than two) which generated a minimum
    distance. (If there is a pair or set which generated the distance and
    includes the home, select it.)
  - Select one in the pair (or set) as the surviving home agent. (Select the
    home if it is in the pair or set.)
  - Update the corresponding maximum shared distance.
    if  ($P$ is not in the pair (or set))
      - Set $P$ to a vacated pointer field.
  - Invalidate still vacated pointer fields.
}
```

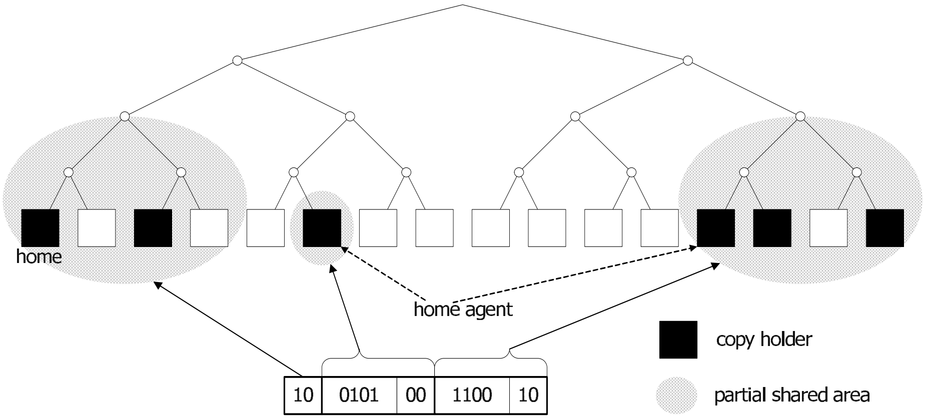


Fig. 4. Adaptive hierarchical coarse directory.

This scheme indicates the locations of copies accurately when the number of copies is not more than N. When more than N, the shared area consists of $N+1$ partial shared areas. From the point of view of adaptability to copy creation and utilization of maximum shared distances, we call the directory "*adaptive hierarchical coarse directory*". Figure 4 illustrates the directory when N is two. In the figure, a home and two home agents form three partial shared areas.

The size of the adaptive hierarchical coarse directory depends on the number of pointer fields. The size of the pseudo-full-map directory is $n \times m$ in a n-ary tree network the height of which is m. This size equals that of two pointers when n is two or four. From this point, the adaptive hierarchical coarse directory which has two pointer fields is almost equal to the pseudo-full-map in the size. Here, the directory includes the other three fields that hold each maximum shared distance. The minimum width of the field depends on n and N. For example, on a quad tree network which has 65,536 processors, the pseudo-full-map needs $4 \times \log_4 65536 = 32$ bits. On the other hand, the adaptive hierarchical coarse directory with $N = 2$ needs $2 \times \log_2 65536 + 3 \times \log_2 \log_4 65536 = 41$ bits.

When N is not large, it is really possible to perform multicasting and combining by adding the directory to a packet. Although it is desirable to give at least $N = n - 1$ pointer fields on a n-ary network system for preventing n copies from causing broadcasting over the system in the worst case, it is important to keep a balance between the amount of memory for directories and the performance improvement.

5 Evaluation of Scalability

In this section, we compare the adaptive hierarchical coarse directory with other directories in terms of the size, latency and traffic in coherence transactions.

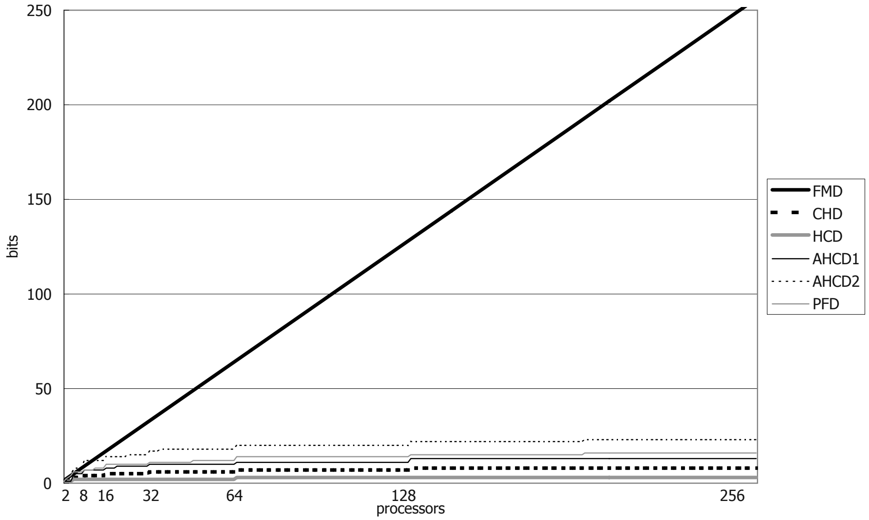


Fig. 5. Directory size.

5.1 Directory Size

The size of an adaptive hierarchical coarse directory is compared with those of other directories; a full-map directory, chained directory, hierarchical coarse directory and pseudo-full-map directory. Figure 5 shows the directory sizes per memory block when the network is a quad tree and the number of processors is from 2 to 256. In the figure, The horizontal axis indicates the number of processors, and the vertical axis indicates the bit width of the directories. “FMD”, “CHD”, “HCD” and “PFD” mean a full-map directory, chained directory, hierarchical coarse directory and pseudo-full-map directory, respectively. “AHCD1” and “AHCD2” means the adaptive hierarchical coarse directory with $N = 2$ and $N = 3$, respectively. Practically, the size of the chained directory per memory block depends on the number of existing copies. Here, the size with no copies, that is the size of a pointer, is given in CHD.

The figure indicates that the width of FMD is proportional to the number of processors and that of PFD is proportional to the logarithm of the number of processors, that is proportional to the height of the tree network. The width of HCD is proportional to the logarithm of the logarithm of the number of processors ($\log_2 \log_4 n$, n is the number of processors) and smaller than that of the other directories. The width of AHCD1 is a little smaller than that of PFD. Although AHCD2 is larger than PFD, the size is sufficiently small for large-scale systems.

5.2 Time Required for Coherence Transaction

From the actual implementation of the hardware DSM system on the prototype machine [1], various values were obtained, such as the time required for a message

Table 1. Parameters in each programs.

Program	Parameters
FFT	65,536 complex double
LU	256×256 matrix 32 by 32 element blocks
OCEAN	Grid size; 130×130 Grid resolution; 20,000 Time between relaxations; 28,800
WATER-SPATIAL	Number of molecules; 4,096
WATER-NSQUARED	Number of molecules; 512

Table 2. The mean number of copies in coherence transaction.

Program	Mean number of copies	
	invalidate	update
FFT	3.14	25.94
LU	1.88	6.11
OCEAN	2.51	15.42
WATER-SPATIAL	2.04	6.03
WATER-NSQUARED	1.28	65.50

to pass through a switch. Using them, coherence processing (invalidation and update) in a larger scale system is considered, and the adaptive hierarchical coarse directory is compared with other directory schemes; full-map, hierarchical coarse directory and pseudo-full-map directory. The adaptive hierarchical coarse directory, hierarchical coarse directory and pseudo-full-map directory are used with multicasting and combining.

We employed ABSS [11] that is the augmentation-based SPARC simulator to generate traces of memory reference. In the simulation, the size of memory (cache) block is 32 bytes and the number of processors is 256. Then, the trace generated by ABSS is input to a directory simulator that computes the number of cycles taken to complete a coherent transaction for each write to shared blocks. In the directory simulator, the interconnection network is quad tree and the same values as those of the prototype machine are used for the times required by the system elements such as a memory controller, network interface and network switch.

We applied the simulation system to several programs from the SPLASH-2 benchmark suite [12]; FFT, LU, OCEAN, WATER-SPATIAL and WATER-NSQUARED. Table 1 shows values of parameters in the programs we selected. Table 2 shows the mean number of copies that should be invalidated or updated per coherence transaction in execution of these five programs.

Table 3 shows the mean number of cycles per transaction. “FMD”, “HCD”, “AHCD1” and “AHCD2” in the table are the same as section 5.1. “SM”, “LARP” and “LPRA” are the tree methods in pseudo-full-map directory. In

Table 3. The mean number of cycles in coherence transaction.

Program	Protocol	Directory						
		FMD	HCD	AHCD1	AHCD2	SM	LARP	LPRA
FFT	Invalidate	119.0	171.0	163.6	154.1	157.1	154.6	167.9
	Update	423.4	263.9	219.4	176.4	192.6	183.1	249.4
LU	Invalidate	94.4	157.3	138.2	137.9	140.0	137.8	153.4
	Update	302.8	268.4	260.5	245.3	233.7	224.7	262.4
OCEAN	Invalidate	117.0	165.6	156.0	147.8	149.5	148.4	162.1
	Update	264.0	259.7	217.2	183.6	193.6	185.2	244.3
WATER-SPATIAL	Invalidate	103.9	159.2	141.2	139.9	143.2	139.8	155.3
	Update	207.9	266.6	218.5	189.7	228.6	216.3	252.3
WATER-NSQUARED	Invalidate	96.1	167.4	150.0	146.7	147.6	147.3	163.6
	Update	777.3	267.6	258.3	244.6	246.7	242.5	261.4

LU with an update protocol, AHCD2 is a little larger than SM and LARP. However, we confirmed that AHCD with three pointers generated smaller value, 137.9 cycles in the same program. In other cases, AHCD2 roughly generates the smallest or the secondly smallest value. From the results, AHCD with two or more pointers generates small number of cycles on average.

5.3 Network Traffic

We show network traffic during coherence processing generated by the same programs as in section 5.2. The same traces in section 5.2 are used. The directory simulator calculates the number of switch-to-switch packets on all branches in the quad tree interconnection network during a coherence transaction.

Table 4 shows the mean number of packets per transaction. Although AHCD2 is larger than the pseudo-full-map directory in LU with an update protocol, we confirmed that AHCD with three pointers generated 217.2 packets in LU. In other programs, AHCD2 is the smallest or the secondly smallest. From these results, AHCD with more than one pointer generates small number of packets on average.

6 Related Work

There are several directory schemes whose information dynamically changes from complete to incomplete, by reconstructing the directory structure[5].

In the superset scheme [8], pointers directly specify the location of sharing processors when the number of copies does not exceed that of pointers. When an overflow occurs, the directory is represented by a composite pointer that is made out of two pointers. Each field of the composite pointer is in one of three states: 0, 1, or X (both), and is thus composed of two bits. When a processor

[5] Pseudo-full-map directory is originally of this type, which switches from a full-bit vector to LPRA [9].

Table 4. The mean number of packets in coherence transaction.

Program	Protocol	Directory						
		FMD	HCD	AHCD1	AHCD2	SM	LARP	LPRA
FFT	Invalidate	45.4	654.5	379.2	76.6	137.6	199.0	255.3
	Update	381.4	643.9	366.5	100.6	163.3	232.6	279.5
LU	Invalidate	20.4	531.3	37.8	17.3	42.4	143.3	144.7
	Update	91.5	676.2	620.2	459.4	335.4	344.0	431.4
OCEAN	Invalidate	36.9	595.3	213.4	38.5	85.4	172.8	197.4
	Update	226.6	613.9	306.7	123.6	160.5	224.6	250.4
WATER-SPATIAL	Invalidate	25.5	531.1	120.3	22.9	68.9	142.3	154.7
	Update	84.3	638.8	321.2	153.2	189.0	214.5	240.9
WATER-NSQUARED	Invalidate	19.3	610.0	107.7	17.1	55.3	165.1	180.7
	Update	950.9	668.1	583.3	489.0	485.2	489.3	499.8

joins sharing members, the processor number is compared with the pointer, and bit fields which differ each other are set to X. This scheme makes a superset of the processors that have a copy, and can point to 2^k copies at maximum by using two pointers length of which is k. During a coherence transaction, a coherence message is sent to all processors that correspond to any bit pattern obtained by replacing each X field with 0 or 1. Hence, a processor which does not hold a copy might receive the coherence messages. This leads to redundant communications. This scheme generates traffic depending on the location of sharing processors, not on the number of copies. In the worst case, sharing by two processors can cause broadcasting for coherence processing. For example, when processors whose number is "0000" and "1111" share a block, this is the case.

In the coarse vector scheme [13], when an overflow of sharing occurs, processors are grouped and a bit is assigned to each group. Groups are then identified by the same way as a full-map scheme. The number of processors that the scheme can cover is k times as large as the bit width of the directory where k is the size of each group. All groups have the same fixed size. Here, redundant coherence messages might be sent as in the superset scheme since all processors in any group in which at least one processor has a copy are regarded as a copy holder. On the other hand, the adaptive hierarchical coarse directory can have groups (partial shared areas) whose size differs and is variable. The size changes according to the number of sharing processors in the neighborhood.

In the segment directory [14], after an overflow, a pointer field changes into the structure which consists of a segment vector and a segment pointer. A segment is a part of all processors and contains consecutive K processors. Therefore, when the number of processors is N, there are N/K segments. The segment vector is a bit vector within a segment and has K bits. The segment pointer indicates which segment the segment vector is applied to. The size of the segment pointer is $log_2 N/K$. In this scheme, the size of a group (segment) is fixed. When the size is large, it requires not a small number of bits for the segment vector. Since this directory makes complete sharing information for a memory block, the same number of directory elements as that of segments are needed to cover all processors. Each segment directory element is dynamically generated as the need arises. When sharing processors are scattered, many directory elements are generated and the total size of the directory for a memory block exceeds that of full-map directory, which means that this structure is not suitable for an update protocol.

Multilayer clustering [15] uses the level of the root of the minimum subtree that includes all the sharers as the sharing information. This technique is the same as the hierarchical coarse directory we proposed in [1]. Further, the multilayer clustering dynamically organizes several subtrees. In this scheme, only symmetric nodes of a home processor can become a home agent that forms a subtree. On the other hand, any node can be a home agent in the adaptive hierarchical coarse directory.

7 Conclusion

In this paper, we described scalability issues of directory schemes in DSM systems and showed that the hierarchical coarse directory provided good scalability in terms of directory size compared to other directories. However, there is a possibility that the directory generates more traffic than others when the number of sharing processors is not large and the copies are scattered.

We proposed a directory scheme, "*adaptive hierarchical coarse directory*" to alleviate the increase of network packets caused by scattered copies. When the number of copies is small, particularly in an invalidation protocol, the directory functions as a limited directory. On the other hand, when an update protocol increases copies, it adaptively makes up sharing information which consists of several partial shared areas.

The effectiveness of the adaptive hierarchical coarse directory was evaluated by using SPLASH-2 programs. This directory with two pointers exhibited the small number of cycles and packets required for processing an invalidation transaction. On the other hand, with an update protocol, it generated the small value for four programs. For LU, the organization of two pointers was not adequate to alleviate the time and traffic. However, three pointers well reduced them. The characteristics of the directory can decrease an obstacle to other local processing in a multitasking/multiuser environment when threads of a process are not locally allocated by an operating system or when false sharing occurs.

References

1. Tanaka, K., Matsumoto, T., Hiraki, K.: Lightweight Hardware Distributed Shared Memory Supported by Generalized Combining. *Proc. of 5th International Symposium on High-Performance Computer Architecture (HPCA)*, pp. 90–99, Jan 1999.
2. Tanaka, K., Matsumoto, T., Hiraki, K.: On Scalability Issue of Directory Schemes of Hardware Distributed Shared memory. *9th Workshop on Scalable Shared Memory Multiprocessors (SSMM)*, Jun 2000.
3. Censier, L.M., Feautrier, P.: A New Solution to Coherence Problems in Multicache Systems. *IEEE Transactions on Computers*, C-27(12), pp. 1112–1118, Dec 1978.

4. Chaiken, D., Kubiatowicz, J., Agarwal, A.: LimitLESS Directories: A Scalable Cache Coherence Scheme. *Proc. of 4th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS–IV)*. pp. 224–234, Apr 1991.
5. James, D., Laundrie, A.T., Gjessing, S., Sohi, G.S.: Distibuted–Directory Scheme: Scalable Coherent Interface. *Computer*, 23(6), pp. 74–77, Jun 1990.
6. Thapar, M., Delagi, B.: Distributed–Directory Scheme: Stanford Distributed Directory Protocol. *Computer*, 23(6), pp. 78–80, Jun 1990.
7. Hagersten, E., Landin, A., Haridi, S.: DDM–A Cache-Only Memory Architecture. *Computer*, 25(9), pp. 44–54, Sep 1992.
8. Agarwal, A., Simoni, R., Hennessy, J., Horowitz, M.: An Evaluation of Directory Schemes for Cache Coherence. *Proc. of 15th International Symposium on Computer Architecture (ISCA)*, pp. 280–289, Jun 1988.
9. Matsumoto, T., Hiraki, K.: A Shared Memory Architecture for Massively Parallel Computer Systems. *IEICE Japan SIG Reports*, 92(173), pp. 47–55, Aug 1992. (In Japanese)
10. Matsumoto, T., Nishimura, K., Kudoh, T., Hiraki, K., Amano, H., Tanaka, H.: Distributed Shared Memory Architecture for JUMP-1: a General-Purpose MPP Prototype. *Proc. of International Symposium on Parallel Architectures, Algorithms and Networks (ISPAN)*, pp. 131–137, Jun 1996.
11. Sunada, D., Glasco, D., Flynn, M.: ABSS v2.0: a SPARC Simulator. *Proc. of the 8th Workshop on Synthesis And System Integration of Mixed Technologies (SASIMI '98)*, Oct 1998.
12. Woo, S.C., Ohara, M., Torrie, E., Singh, J.P., Gupta, A.: The SPLASH-2 Programs: Characterization and Methodological Considerations. *Proc. of 22th International Symposium on Computer Architecture (ISCA)*, pp. 24–36, Jun 1995.
13. Gupta, A., Weber, W., Mowry, T.: Reducing Memory and Traffic Requirements for Scalable Directory-Based Cache Coherence Schemes. *Proc. of International Conference on Parallel Processing (ICPP)*, pp. I-312–321, Aug 1990.
14. Choi, J.H., Park, K.H.: Segment Directory Enhancing the Limited Directory Cache Coherence Schemes. *Proc. of 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing (IPPS/SPDP)*, pp. 258–267, Apr 1999.
15. Acacio, M.E., Gonzalez, J., Carcia, J.M., Duato, J.: A New Scalable Directory Architecture for Large-Scale Multiprocessors. *Proc. of 7th International Symposium on High-Performance Computer Architecture (HPCA)*, pp. 97–106, Jan 2001.

A Compiler-Assisted On-Chip Assigned-Signature Control Flow Checking[*]

Xiaobin Li and Jean-Luc Gaudiot

Department of Electrical Engineering and Computer Science
University of California, Irvine
{xiaobinl, gaudiot}@uci.edu
http://pascal.eng.uci.edu

Abstract. As device sizes continue shrinking, lower charges are needed to activate gates, and consequently ever smaller external events (such as single ionizing particles of naturally occurring radiation) will be able to upset the correct functioning of complex modern microprocessors. Therefore, designers of future processors must take this new fact into account and should incorporate in their design fault-tolerant features which will allow processors to continue operating correctly even when such faults have occurred. Many faulty conditions are control flow errors which cause processors to violate the correct sequencing of instructions. Indeed, they amount to between 33% and 77% of all run-time errors. We present here a new compile-time signature assignment algorithm (the signature checking technique is a well-known approach to detect control flow errors). We also present the theoretical proof as well as the fault detection coverage analysis of our algorithm. We then describe the required enhancement to the basic microarchitecture: an on-chip assigned-signature checker which is capable of executing three additional instructions (SIC, SIJ, SIJC). This allows the processor to efficiently check the run-time sequence and detect control flow errors.

1 Introduction

As computer systems have become irreplaceable tools of modern society, with the benefits these systems bring to us comes a great potential for harm when they fail to perform their functions or perform them incorrectly. This is further exacerbated by new technologies of integration as the number of transistors and the clock rate of processors have shown an exponential growth rate [1]. However, smaller device sizes, reduced voltage levels, and higher transistor counts correspondingly raise concerns of higher transient faults rates. For one thing, radiation-induced soft errors are predicted to become increasingly significant in the near future [2,3,4]. In order to handle these inevitable errors, we must integrate in our design fault-tolerant features so that the processor can continue to correctly perform its specified tasks despite the occurrence of logic errors [5]. Such designs as Itanium [6], IBM Power4 [7], Fujitsu SPARC64 [8], etc., already include transient fault detection and recovery mechanisms.

[*] This paper is based upon work supported in part by NSF grants CCR-0234444 and INT-0223647. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

P.-C. Yew and J. Xue (Eds.): ACSAC 2004, LNCS 3189, pp. 554–567, 2004.

We concentrate here on protecting against *control flow errors* (those which cause a processor to violate the correct sequencing of instructions). Indeed, abstractions of program execution behavior can be formed based on various considerations which include control flow, memory access, I/O, and object type or range [9]. The cause of control flow errors could be the failure of any one of a variety of microarchitectural components such as instruction cache, program counter operation, branch unit, etc. Indeed, it has been found that these control flow errors account for between 33% [10] and 77% [11] of all run-time errors.

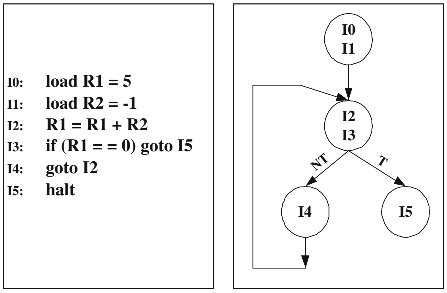
Signature checking is a well-known technique used to detect control flow errors [9, 11,12,13,14,15,16,17,18,19]. It can be implemented as either assigned-signature control flow checking or derived-signature control flow checking. In this paper, we focus on the former because it could offer better fault detection coverage. At compile time, we assign to each basic block a signature, and then at run time, an on-chip checker executes three additional signature checking instructions in order to check run time-computed signatures against assigned signatures. Any discrepancy indicates that an error occurred.

The goal of this paper is to describe the algorithm which protects against run-time control flow errors and its simple implementation. In Section 2, we introduce the principles of signature checking. The compile time signature assignment algorithm is outlined in Section 3. An on-chip checker with the ability to execute three signature checking instructions and its possible implementation are described in Section 4. These three instructions are proposed additions to a conventional instruction set. Conclusions are presented in Section 5.

2 The Concept of Signature Checking

Signature control flow checking techniques are used to monitor the program execution sequence in order to determine if a legal control flow is being followed. Various signature checking techniques have been proposed in the past [9,11,12,13,14,15,16,17,18,19]. Basically, there are **two** phases of signature checking: compile-time signature generation and run-time signature validation.

In the back end stages, in order to express the program control flow, compilers usually build a control flow graph (CFG), in which a *node* or a basic block is a sequence of instructions with no branch-in except for the entry point and no branch-out except for the exit point and *directed edges* are used to represent jumps in the program control flow [20]. Fig. 1 illustrates this concept by a simple example. Thus, in the first phase of signature checking, which is based upon the CFG, the compiler pre-computes the signatures associated with each node of the CFG, and then either embeds signatures into the original codes [9,11,13,14,16,19] or provides that information directly to the watchdog [15,18]. At this point, we could have two techniques for pre-computing signatures: the first, assigned-signature control flow checking [15,16,19], associates with each node an arbitrary signature, for example, a prime number. Conversely, the second technique, derived-signature control flow checking [9,11,13,14,18], derives signatures from the nodes themselves, for example by deriving a checksum from the binary code of the instruction inside a node and then using that checksum as the signature.



```
I0:   load R1 = 5
I1:   load R2 = -1
I2:   R1 = R1 + R2
I3:   if (R1 = = 0) goto I5
I4:   goto I2
I5:   halt
```

Fig. 1. An example program and its CFG

During the second phase, the checking engine, which can be either the watchdog or the host CPU, computes run time signatures and then check them against the compile time pre-computed signatures. If the signatures differ, it means that an error occurred.

Although the second phase of the assigned-signature checking algorithm and the derived-signature checking algorithm are essentially the same, assigned signature checking techniques have two major drawbacks: the need for registers to hold signatures and the performance overhead due to the need to execute extra instructions related to the assigned-signature checking [9,19]. For example, in [19], the overhead in terms of code size ranges from 26.6% to 61.9% while the overhead in terms of execution time ranges from 16.2% to 58.1%. Conversely, derived signature checking techniques require a signature generator/checker circuit to process the signature and might not guarantee that each node has a unique signature, which might consequently impact the fault coverage.

3 The Compiler-Assisted Signature Assignment Algorithm

The assigned-signature checking technique is based on a comparison between the compiler assigned signature with the one calculated at run time. Any difference between these two signatures indicates that a control flow error has occurred. To address the performance overhead associated with assigned-signature checking, we use additional hardware to trade this off, as will be explained in Section 4.

3.1 The Control Flow Checking Algorithm

Compiler time assigned reference signature (S). As discussed before, the program control flow can be expressed as a CFG. We start with a given node V_i of the CFG, and assign to it a **unique** number, which is called the *state code* of the node[1]. This code is denoted by *D(i)*. Then, we compute the reference signature *S(i)* of this node by using the following formula:

$$S(i) = D(i) \oplus D(pred(V_i)) \ . \tag{1}$$

[1] A simple way to assign unique state codes to nodes may be to number each node of the CFG in sequence, as shown in Fig. 7.

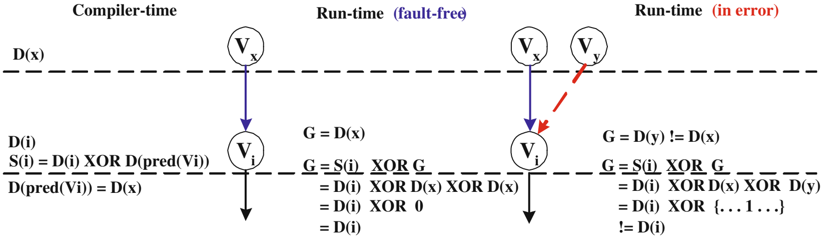


Fig. 2. Proof of the control flow checking algorithm

where $pred(V_i)$ is the immediate predecessor of node V_i in the CFG (note that $\oplus$ is an exclusive-OR function). Furthermore, we assume for the moment that each node has only one immediate predecessor. More complex cases will be discussed later.

Run time signature (G). A global register holds the *run time signature* G of the node currently executing. When the program execution changes the control flow to a new node, e.g., V_i, G is updated by the following formula:

$$G = G \oplus S(i) \quad . \tag{2}$$

where *S(i)* is the reference signature of the current new node V_i.

Then, the core of the control flow checking mechanism consists in checking the run time signature G against the static state code *D(i)* (the one assigned by the compiler) as follows[2]:

```
Do   G ⊕ D(i)
BNEZ exception-handler
```

Note that this comparison would take place whenever the run time control flow enters a new node of the CFG.

Proof (of the control flow checking algorithm). Assume (Fig. 2) that, instead of transferring control from V_x to V_i (left side of Fig. 2), we had erroneously entered V_i from V_y (right side of Fig. 2). Further assume that the reference signature of V_i had been assigned by the compiler to be $S(i) = D(x) \oplus D(i)$. The run-time signature G when the control of program is at node V_y is: $G = D(y)$ which would be different from *D(x)* since the state code is *unique* to each node. Then, after entering node V_i, the run time signature would be updated by the following formula:

$$\begin{aligned} G &= G \oplus S(i) \\ &= D(y) \oplus D(x) \oplus D(i) \end{aligned}$$

Since $D(x) \neq D(y)$, some bits of the result from $D(x) \oplus D(y)$ are 1s, as shown by $\{\ldots 1 \ldots\}$ in Fig. 2. Because of these bits which have the value ‘1’ instead of ‘0’, when exclusive-ORed with *D(i)*, they will flip the corresponding bits of *D(i)*. As such, the final result is: $G \neq D(i)$ which means that a fault has been detected. □

[2] BNEZ is equivalent to “branch to target if the result is not equal to zero.” As such, if $G \oplus D(i) \neq 0$, the exception handler is triggered.

Justifying signature (J)—Handling multiple-branch-in nodes. Now, we need to consider the case when a node has multiple immediate predecessors. Indeed, in normally complex CFGs, a node may have multiple immediate predecessors. We would call such a node a *multiple-branch-in* (MBI) node: it is a node whose number of immediate predecessors is greater than one. To simplify the discussion, we denote the set of *pred(MBI)* as[3]:

$$\$ = \left\{ V_k \mid V_k \text{ is an immediate predecessor of MBI} \right\} \ . \tag{3}$$

When dealing with such MBI nodes, as required in (1), we must choose one of the immediate predecessors as the *primary immediate predecessor* (or primary node, for short). Also, since there is more than one path *up* from an MBI node, we associate with each immediate predecessor an additional parameter which we call the *justifying signature*. The justifying signature is used at run time to verify that all immediate predecessors to the MBI node are *legal* antecedents to that node.

The following outlines the *compile-time* MBI node handling algorithm:

1. Arbitrarily select a node from $\$$ as the MBI node primary node (assume V_j for the rest of this discussion). Note that we leave the discussion of primary node selection later.
2. The reference signature of the MBI node is governed by the selected primary node V_j as:
$$S(MBI) = D(MBI) \oplus D(j) \ . \tag{4}$$
3. For every node $V_k \in \$$, associate it with the justifying signature given by the following formula:
$$J(k) = D(k) \oplus D(j) \ . \tag{5}$$
Note that now each node $V_k \in \$$ has **two** signatures: the reference signature *S(k)* and the justifying signature *J(k)* as illustrated in Fig. 3 where V_7, V_8 and V_9 are the MBI nodes[4].

The *run-time* MBI node handling algorithm could be described as follows:

1. We denote control flow changes as: $V_k \rightarrow$ MBI. First, the run time signature is updated according to the following formula:
$$G = G \oplus S(MBI) \oplus J(k) \ . \tag{6}$$
2. Finally, the MBI node run-time control flow checking can be applied as discussed before:

```
Do   G ⊕ D(MBI)
BNEZ exception-handler
```

3.2 The Fault Detection Coverage of the MBI Node Handling Algorithm

Consider a general case of MBI node control flow change: $V_k \rightarrow$ MBI. According to the relationship between the node V_k and the node "MBI," there are three possible cases:

[3] Then the definition of an MBI node can be given as the cardinality of $\$$, i.e., the number of elements in the set $\$$, is greater than one: $|\$| > 1$.

[4] We use doubly circled nodes to represent primary nodes.

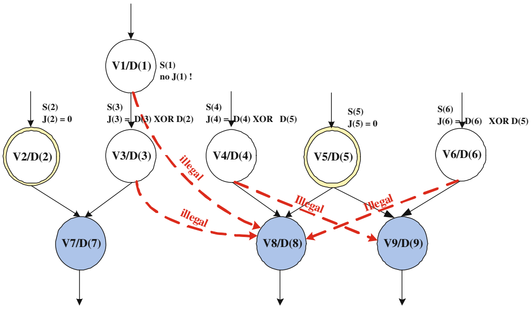


Fig. 3. Analysis of the MBI node handling algorithm

1. If $V_k \in \mathbb{S}$, which means that the control flow change is legal, as discussed before, we can easily prove that the updated run time signature is: $G = D(MBI)$.
2. If $V_k \notin \mathbb{S}$, which means that the control flow change is illegal, two cases must be separately considered:
   a) V_k is an immediate predecessor of *another* MBI node, which means that *J(K)* has been defined;
   b) V_k is not an immediate predecessor of *any* MBI node. Then, *J(k)* is null at compile-time and then without loss of generality, *J(k)* will be a random number at run-time (whatever is left in the corresponding storage cell at that time).

Justifying signature has been defined. Consider the control flow change: $V_i \rightarrow V_j$ where V_j is an MBI node and its set of $pred(V_j)$ is $\mathbb{S}_j$. However, $V_i \notin \mathbb{S}_j$, i.e., the control flow change is illegal, V_i is instead an immediate predecessor of another MBI node, say V_m, hence $V_i \in \mathbb{S}_m$. Moreover, V_x has been selected as the primary node of V_j and V_y as the primary node of V_m. Then, when entering V_j, the run time signature is updated by using the following formula:

$$\begin{aligned} G &= G \oplus S(j) \oplus J(i) \\ &= D(i) \oplus D(j) \oplus D(x) \oplus D(i) \oplus D(y) \\ &= D(j) \oplus D(x) \oplus D(y) \end{aligned}$$

Therefore, two cases need to be considered:

1. As long as $D(x) \oplus D(y) = 0$, we end up with $G = D(j)$, which means that a control flow error has escaped detection. The faulty condition $D(x) \oplus D(y) = 0$ is satisfied only if $D(x) = D(y)$, i.e., node V_x is the same as node V_y (remember that the state code of each node is unique). Examples of illegal control flow changes such as $V_6 \rightarrow V_8$ and $V_4 \rightarrow V_9$, are shown in Fig. 3. In both cases, two MBI nodes V_8 and V_9 share node V_5 as their primary node.

Observation 1. *When two MBI nodes, V_j and V_m, share their primary node, V_x (= V_y), any illegal control flow change: $V_i(\in \mathbb{S}_m$ and $\notin \mathbb{S}_j) \rightarrow V_j$ and any illegal control flow change: $V_l(\in \mathbb{S}_j$ and $\notin \mathbb{S}_m) \rightarrow V_m$ cannot be detected.*

Further, we denote the probability of these illegal control flow changes as $\mathrm{P_{ND1}}$.

2. On the other hand, if $V_x \neq V_y$, i.e., **no** primary node sharing, we have: $G \neq D(j)$ which means that the control flow error can be successfully detected. An example for this case is shown in Fig. 3 as the illegal control flow change: $V_3 \rightarrow V_8$.
   To summary the above two cases, we can state the following:

   Observation 2. *The fault detection coverage may decrease if two MBI nodes share the primary node. In another words, if a node V_x has multiple branch-outs, for example, the exit statement of the node is a conditional branch, and if more than two (including two) branch destination nodes are MBI nodes, the node V_x should not be selected as a primary node.*

Justifying signature is random. Consider the control flow change: $V_i \rightarrow V_j$ where V_j is an MBI node and its set of $pred(V_j) = \$_j$. Further assume that V_x has been selected as the primary node of V_j. However, $V_i \notin \$_j$, i.e., the control flow change $V_i \rightarrow V_j$ is illegal. Also, V_i is not an immediate predecessor of any MBI node such that *J(i)* has not been defined and we deal with it as a random number. An example of an illegal control flow change: $V_1 \rightarrow V_8$ is shown in Fig. 3. In this case, when entering V_j, the run-time signature is updated by using the following formula:

$$\begin{aligned} G &= G \oplus S(j) \oplus J(i) \\ &= D(i) \oplus D(j) \oplus D(x) \oplus J(i) \end{aligned}$$

Because of the randomness of *J(i)*, two cases must be considered:

1. If $D(i) \oplus D(x) \oplus J(i) = 0$, we have $G = D(j)$ which means that the control flow error escapes detection;
2. If $D(i) \oplus D(x) \oplus J(i) \neq 0$, we have $G \neq D(j)$ which means that the algorithm has successfully detected the control flow error.

Fortunately, the probability for $D(i) \oplus D(x) \oplus J(i)$ to be zero is very low: it can happen only if $J(i) = D(i) \oplus D(x)$. Given the n-bit size of state codes and signatures, the probability is:

$$\mathrm{P_{ND2}} = \mathrm{P}\left\{J(i) = D(i) \oplus D(x)\right\} = 2^{-n} \quad . \tag{7}$$

In summary, the fault detection coverage of the MBI node handling algorithm is:

$$\begin{aligned} \mathrm{C} &\equiv \mathrm{P}\left\{\text{fault detection} \middle| \text{fault existence}\right\} && (8) \\ &= \mathrm{P}\left\{\text{control flow error detection} \middle| V_k \rightarrow \text{MBI and } V_k \notin \$\right\} && (9) \\ &= 1 - \mathrm{P_{ND1}} - \mathrm{P_{ND2}} \quad . && (10) \end{aligned}$$

3.3 The If-Then-Else Node Handling Algorithm

As mentioned by Oh et al. in [19], primary nodes are randomly selected which would contradict our Observation 2. Furthermore, randomly selecting the primary node may

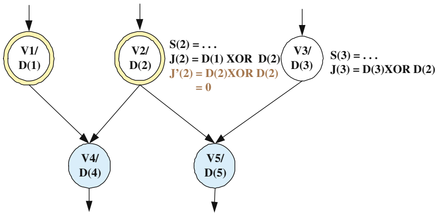


Fig. 4. An example of ITE node with two justifying signatures

result in conflicts as illustrated in Fig. 4. Indeed, if V_1 had been selected as the primary node for V_4 and V_2 for V_5, respectively, we would have to create two justifying signatures for node V_2: as far as MBI node V_4 is concerned, the justifying signature of V_2 is: $J(2) = D(1) \oplus D(2)$; whereas as far as MBI node V_5 is concerned, the justifying signature of V_2 is: $J'(2) = D(2) \oplus D(2) = 0$.

Hence, for the control flow change: $V_2 \rightarrow V_4$, *J(2)* should be used to update the run-time signature whereas for the control flow change : $V_2 \rightarrow V_5$, only *J'(2)* is the correct choice. Anything corrupted up to this level could result in faulty control flow error detection. Simply speaking, for the **legal** control flow change: $V_2 \rightarrow V_4$, if the justifying signature *J'(2)* had been used to update the run time signature, we would end up with $G \neq D(4)$ such that a control flow error could be flagged, a false alarm. Unfortunately, such situations[5] have not been addressed in [19].

The necessary conditions for a node associated with two[6] justifying signatures are:

1. The exit of the node is a conditional branch, that is to say the node is an if-then-else (ITE) node;
2. Both branch destinations are MBI nodes.

In short, we need to distinguish the two justifying signatures: one for the then-branch flow (the resolved branch condition is not-taken), the other for the else-branch flow (the resolved branch condition is taken).

Figure 5 shows a hardware-based algorithm[7]: at **compile time**, when an MBI node traces back its immediate predecessors for the purpose of justifying signatures, the associated directed edges are checked (directed edges are given by the CFG): if the edge is a "taken" path, the associated justifying signature will be placed into the TJ register; whereas if it is a "not-taken" path, the NTJ register is used for the associated justifying

[5] These situations are not rare: conditional branches are extremely common in regular programs. From the following discussions, we will see that a node with a conditional branch could be associated with two justifying signatures.

[6] If switch statements are allowed, that more than two justifying signatures are associated with a node is possible. However, we assume that the compiler has converted all switch statements into the equivalent if-then-else constructs, as presented in [14].

[7] We have not, in this work, considered checking the flow of conditional branches. More specifically, refer to Fig. 5, the case when a transient fault causes the ITE node to branch to the else_node incorrectly whereas it should have branched to the then_node, has not been considered.

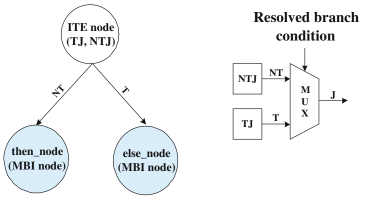


Fig. 5. An hardware approach for ITE node with two justifying signatures (T = taken; NT = not-taken; TJ = justifying signature for ITE_node → else_node; NTJ = justifying signature for ITE_node → then_node)

signature. At **run time**, the resolved branch condition is used to select the appropriate justifying signature for updating the run time signature. More details will be given in Section 4.

4 Hardware Enhancement for Control Flow Checking

As discussed before, the assigned-signature checking technique has an inherent performance overhead drawback. However, with advances in CMOS technology, we have an abundance of cheap hardware resources [1]. Moreover, our proposed mechanism can be simply implemented in any modern microprocessor at little additional cost. Hence, in this section, we first introduce three additional instructions dedicated to control flow checking, and then design a simple hardware implementation to execute these instructions. We will also provide a comprehensive control-flow checking algorithm based on these hardware enhancements. In the end, we will show the benefit from trading hardware off a reduction in performance overhead.

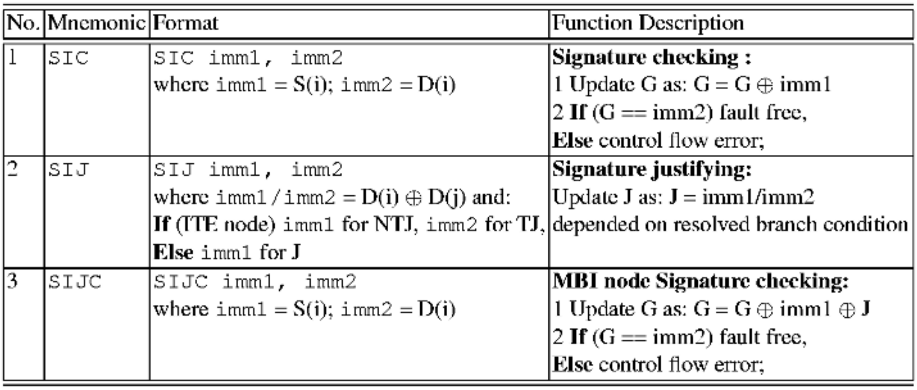
4.1 Additional Instructions

The three additional instructions dedicated to the assigned-signature control flow checking are succinctly described in Table 1. Instruction `SIC` is used to check for control flow errors in non-MBI nodes. Instruction `SIJ` is dedicated to signature justification. Instruction `SIJC` is used to check for control flow errors in MBI nodes. The compiler is responsible for the insertion of these additional instructions into the original program so as to achieve run-time control flow checking. The detailed algorithm will be presented in section 4.3.

4.2 Implementation of Additional Instructions — On-Chip Control Flow Checker

A simple on-chip control flow checker to execute the above three additional instructions can be easily designed. Assume a simple five-stage pipeline: Fetch → Decode → Execution → Memory access → Write back. Our on-chip control flow checker would be located in the "Decode" and "Execution" stages.

Table 1. Three additional instructions specification



No.	Mnemonic	Format	Function Description
1	`SIC`	`SIC imm1, imm2` where `imm1` = S(i); `imm2` = D(i)	**Signature checking :** 1 Update G as: G = G $\oplus$ imm1 2 **If** (G == imm2) fault free, **Else** control flow error;
2	`SIJ`	`SIJ imm1, imm2` where `imm1` / `imm2` = D(i) $\oplus$ D(j) and: **If** (ITE node) `imm1` for NTJ, `imm2` for TJ, **Else** `imm1` for J	**Signature justifying:** Update J as: J = imm1/imm2 depended on resolved branch condition
3	`SIJC`	`SIJC imm1, imm2` where `imm1` = S(i); `imm2` = D(i)	**MBI node Signature checking:** 1 Update G as: G = G $\oplus$ imm1 $\oplus$ **J** 2 **If** (G == imm2) fault free, **Else** control flow error;

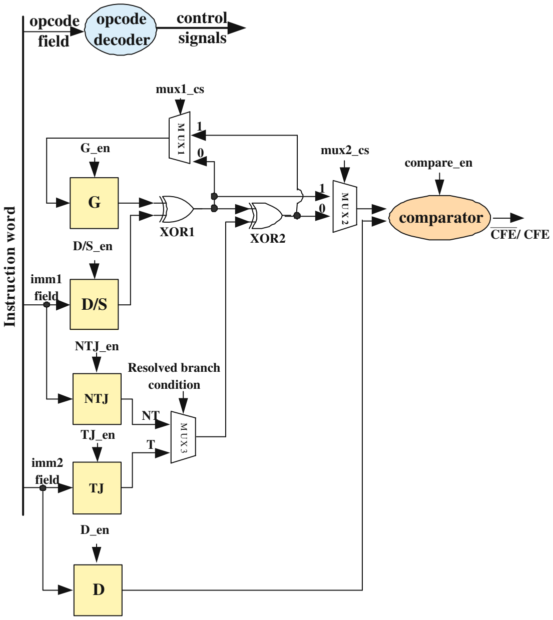


Fig. 6. On-chip control flow checker block diagram

As seen in Fig. 6, a total of five registers are needed to hold the necessary information: register **G** is used for the run time signature; registers **D/S** and **NTJ** receive immediate values from the `imm1` field of their instruction words and registers **D** and **TJ** receive immediate values from the `imm2` field of the instruction words. For each instruction, Table 2 shows the control signals generated by the opcode decoder (also shown in Fig. 6).

Table 2. On-chip control flow checker control signals (en = enable; $\overline{en}$ = disable; X = don't care; BRU = branch unit)

Instr.	G_en	D/S_en	D_en	NTJ_en	TJ_en	mux1_cs	mux2_cs	br. cond.	compare_en
SIC	en	en	en	$\overline{en}$	$\overline{en}$	0	1	$\overline{en}$	en
SIJ	$\overline{en}$	$\overline{en}$	$\overline{en}$	en	en	X	X	$\overline{en}$	$\overline{en}$
SIJC	en	en	en	$\overline{en}$	$\overline{en}$	1	0	from BRU	en

Operation of `SIC` *instructions.* When an instruction word `SIC imm1, imm2` is decoded, its opcode field is fed into the opcode decoder. The decoder then generates the control signals specified in Table 2. The `imm1` filed is received by the enabled register D/S (D/S_en = enable and NTJ_en = $\overline{enable}$) while the `imm2` field is received by the enabled register D (D_en = enable and TJ_en = $\overline{enable}$).

The content of register D/S goes into XOR1 along with the content of register G (G_en = enable). The result is selected by mux2 (mux2_cs = 1). Now the enabled comparator compares the two inputs which are received from mux2 and register D. These have performed the run-time control flow checking. Also, we can see the result of XOR1 is sent to modify register G since we have mux1_cs = 0 and G_en = enable.

Operation of `SIJ` *instructions.* When an instruction word `SIJ imm1, imm2` is decoded, its opcode field is fed into the opcode decoder. The decoder then generates the control signals specified in Table 2. The `imm1` filed is received by the enabled register NTJ (NTJ_en = enable and D/S_en = $\overline{enable}$) while the `imm2` field is received by the enabled register TJ (TJ_en = enable and D_en = $\overline{enable}$).

Operation of `SIJC` *instructions.* When an instruction word `SIJC imm1, imm2` is decoded, its opcode field is fed into the opcode decoder. The decoder then generates the control signals specified in Table 2. The `imm1` filed is received by the enabled register D/S (D/S_en = enable and NTJ_en = $\overline{enable}$) while the `imm2` field is received by the enabled register D (D_en = enable and TJ_en = $\overline{enable}$).

The content of register D/S goes into XOR1 along with the content of register G (G_en = enable). Based on the resolved branch condition, either the content of register NTJ or that of register TJ XOR2 with the result of XOR1. Once again, if no conditional branch result from the branch unit, i.e., not an ITE node, the default "resolved branch condition = NT" such that the content of register NTJ is selected at this point. MUX2 selects the result of XOR2 (mux2_cs = 0). Now the enabled comparator compares the two inputs which are received from mux2 and register D. These have performed the run-time control flow checking. Furthermore, we can see that the result of XOR2 is sent to modify register G since we have mux1_cs = 1 and G_en = enable.

4.3 Using Additional Instructions

To summarize the above discussion, Algorithm 1 shows a comprehensive signature assignment algorithm based on our hardware enhancement instructions. Returning to the example of Fig. 1, our compiler algorithm would produce the modified diagram

Algorithm 1 Embedding three additional instructions into programs

```
Signature-Assignment(program)
Derive CFG of the given program
Assuming we have nodes: $V_i$ (i = 1,2,3,...,N) where N is the total number of nodes in the CFG
Assign a unique state code $D(i)$ to every node $V_i$
for every node $V_i$ in the CFG do
  if $V_i$ is not an MBI node then
    Compute its assigned reference signature as: $S(i) = D(i) \oplus D(pred(V_i))$;
    Place the instruction "SIC imm1, imm2"
    at the beginning of node $V_i$ and before the SIJ instruction, if any
    Assign the values of imm1 and imm2 as: imm1 = S(i) and imm2 = D(i)
  else
    Select a primary node from: $\mathbb{S}$ = the set of pred($V_i$) (assume $V_j$ is selected)
    Assign the reference signature of $V_i$ as: $S(i) = D(i) \oplus D(j)$
    Place the instruction "SIJC imm1, imm2"
    at the beginning of node $V_i$ and before the SIJ instruction, if any
    Assign the values of imm1and imm2 as: imm1 = S(i), imm2 = D(i)
    for every node $V_k \in \mathbb{S}$ (including $V_j$) do
      Place instruction "SIJ imm1, imm2"
      into the node $V_k$ and after the SIC and/or SIJC instructions
      Assign the values of imm1 and imm2 as follows:
      if $V_k \rightarrow V_i$ is a taken path then
        imm1 = X, imm2 = $D(k) \oplus D(j)$
      else if $V_k \rightarrow V_i$ is a not-taken path then
        imm1 = $D(k) \oplus D(j)$, imm2 = X
      else
        imm1 = $D(k) \oplus D(j)$, imm2 = X{$V_k \rightarrow V_i$ is not a conditional branch path}
      end if
    end for
  end if
end for
```

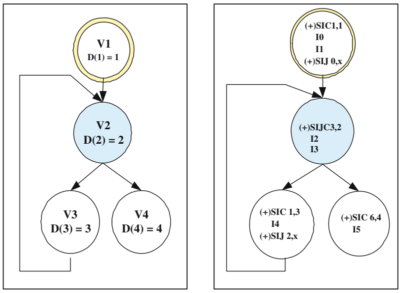


Fig. 7. State code and signature assignment example

shown in Fig. 7. The left-hand side illustrates the state code assignment results and the primary node selection of the MBI node V_2. The right-hand side shows the CFG after insertion of our control flow checking instructions.

Comparing code size overhead. To compare our algorithm with that of Oh et al. in [19], consider a *typical* node consisting of 7 to 8 instructions [1]. In order to check for control flow errors, [19] adds 2 to 4 instructions to each node. The overhead is between 27% and 53%. (As shown in [19], for a number of benchmarks, the code size overhead is between 26.6% and 61.9% whereas the execution time overhead is between 16.2% and 58.1%). Conversely, in our hardware-enhanced approach, the additional instructions are a maximum of 1 or 2 for each node. Therefore, the overhead is only 13% to 27%, which is a significant improvement over [19]. Furthermore, the execution time is given by the following formula [1]:

$$\text{Execution time} = \text{Instr count} \times \text{Clock cycle time} \times \text{Cycles per instr} \ . \qquad (11)$$

With help from our on-chip control flow checker, we could expect a lower execution time than that obtained in [19] when executing the program with the signature checking. This is because we have a smaller instruction count given the same clock cycle time and cycles per instructions.

5 Conclusions

Control flow errors have a high error occurrence ratio relative to other kinds of errors. This is expected to continue increasing as design rules continue decreasing. Signature checking is a well-known and effective technique to detect such errors. We have used this approach to demonstrate our compiler-assisted assignment signature analysis. It includes a compile time algorithm based on a control flow graph which assigns signatures to nodes. We have also designed an on-chip checker for our dedicated instructions used for control flow checking. A comprehensive signature assignment algorithm has also been introduced, and a detailed performance overhead analysis has been presented.

References

1. Hennessy, J.L., Patterson, D.A.: Computer Architecture: A Quantitative Approach. Third edn. Morgan Kaufmann Publishers, Inc. (2002)
2. Borkar, S.: Design Challenges of Technology Scaling. IEEE Micro (1999)
3. Yang, P., Chern, J.H.: Design for Reliability: The Major Challenge for VLSI. Proceedings of the IEEE (1999)
4. Reinhardt, S.K., Mukherjee, S.S.: Transient Fault Detection via Simultaneous Multithreading. In: 27th International Symposium on Computer Architecture. (2000)
5. Hennessy, J.: The Future of Systems Research. IEEE Computer (1999)
6. Quach, N.: High Availability and Reliability in the Itanium Processor. IEEE Micro (2000)
7. Bossen, D.C., Tendler, J.M., Reick, K.: Power4 System Design for High Reliability. IEEE Micro (2002)
8. Ando, H., Yoshida, Y., Inoue, A., Sugiyama, I., Asakawa, T., Morita, K., Muta, T., Motokurumada, T., Okada, S., Yamashita, H., Satsukawa, Y., Konmoto, A., Yamashita, R., Sugiyama, H.: A 1.3-GHz Fifth-Generation SPARC64 Microprocessor. IEEE Journal of Solid-State Circuits (2003)
9. Wilken, K., Shen, J.P.: Continuous signature monitoring: Low-Cost Concurrent-Detection of Processor Control Errors. IEEE Transactions on Computer-Aided Design (1990)

10. Ohlsson, J., Rimen, M., Gunneflo, U.: A Study of the Effects of Transient Fault Injection Into a 32-bit RISC with Built-in Watchdog. In: 29th International Symposium on Fault-Tolerant Computing. (1991)
11. Schuette, M.A., Shen, J.P.: Processor Control Flow Monitoring Using Signatured Instruction Streams. IEEE Transactions on Computers (1987)
12. Mohmood, A., McCluskey, E.J.: Concurrent Error Detection Using Watchdog Processors – A Survey. IEEE Transactions on Computers (1988)
13. Schuette, M.A., Shen, J.P.: Exploiting Instruction-Level Parallelism for Integrated Control-Flow Checking. IEEE Transactions on Computers (1994)
14. Warter, N.J., Hwu, W.M.W.: A Software Based Approach to Achieving Optimal Performance for Signature Control Flow Checking. In: 20th International Symposium on Fault-Tolerant Computing. (1990)
15. Michel, T., Leveugle, R., Saucier, G.: A New Approach to Control Flow Checking without Program Modification. In: 21st International Symposium on Fault-Tolerant Computing. (1991)
16. Alkhalifa, Z., Nair, S., Krishnamurthy, N., Abraham, J.A.: Design and Evaluation of System-Level Checks for On-Line Control Flow Error Detection. IEEE Transactions on Parallel and Distributed Systems (1999)
17. Shirvani, P.P., McCluskey, E.J.: Fault-Tolerant Systems in a Space Environment: The CRC ARGOS Project. Technical Report CRC-TR 98-2, Stanford University (1998)
18. Bagchi, S., Srinivasan, B., Whisnant, K., Kalbarczyk, Z., Iyer, R.K.: Hierarchical Error Detection in a Software Implemented Fault Tolerance (SIFT) Environment. IEEE Transactions on Knowledge and Data Engineering (2000)
19. Oh, N., Shirvani, P.P., McCluskey, E.J.: Control-Flow Checking by Software Signatures. IEEE Transactions on Reliability (2002)
20. Aho, A.V., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools. Addison-Wesley Publishing Company (1986)

A Floating Point Divider Performing IEEE Rounding and Quotient Conversion in Parallel

Woo-Chan Park[1], Tack-Don Han[2], and Sung-Bong Yang[2]

[1] Department of Internet Engineering,
Sejong University, Seoul 143-747, Korea,
pwchan@sejong.ac.kr
[2] Department of Computer Science,
Yonsei University, Seoul 120-749 Korea,
{hantack}@kurene.yonsei.ac.kr
{yang}@cs.yonsei.ac.kr

Abstract. Processing floating point division generally consists of SRT recurrence, quotient conversion, rounding, and normalization steps. In the rounding step, a high speed adder is required for increment operation, increasing the overall execution time. In this paper, a floating point divider performing quotient conversion and rounding in parallel is presented by analyzing the operational characteristics of floating point division. The proposed floating point divider does not require any additional execution time, nor does it need any high speed adder for the rounding step. The proposed divider can execute quotient conversion, rounding, and normalization within one cycle. To support design efficiency, the quotient conversion/rounding unit of the proposed divider can be shared efficiently with the addition/rounding hardware for floating point multiplier.

1 Introduction

An FPU (Floating Point Unit) is a principal component in graphics accelerators [1,2], digital signal processors, and high performance computer systems. As the chip integration density increases due to the advances in semiconductor technology, it has become possible for an FPU to be placed on a single chip together with the integer unit, allowing the FPU to exceed its original supplementary function and becoming a principal element in a CPU [2,3,4,5]. In recent microprocessors, a floating point division unit is built on a chip to speed up the floating point division operation.

In general, the processing flow of the floating point division operation consists of SRT recurrence, quotient conversion, rounding, and normalization steps [6,7,8]. SRT recurrence has been used to perform the division operation for the fraction part and to produce the final quotient and its remainder as in a redundant representation. In the quotient conversion step, the sign bit for the final remainder can be calculated from both the carry part and the sum part of the remainder in a redundant representation. Hence, a conventionally binary represented quotient is produced using the positive part and the negative part of the

P.-C. Yew and J. Xue (Eds.): ACSAC 2004, LNCS 3189, pp. 568–581, 2004.

redundantly represented quotient and the sign bit of the final remainder. After that, the rounding step can be performed using the results from the quotient conversion step. For the rounding step, a high speed adder for increment operation is usually required, increasing the overall execution time and occupying a large amount of chip area.

In some microprocessors, due to design efficiency, the rounding unit for a floating point divider is shared with a rounding hardware for a floating point multiplier or a floating point adder [7,9]. The reasons for this sharing are as follows. First, because the floating point division operation requires many cycles to complete its operation, it does not need to be implemented by a pipeline structure. Second, because the floating point division operation is not used more frequently than other floating point operations, additional hardware for the rounding unit in the floating point divider is not economical. Third, the hardware to support the rounding operation for floating point division can be developed simply by modifying the rounding unit for either a floating point multiplier or a floating point adder. However, this sharing must not affect any critical path of the shared one and should not complicate the control scheme. Therefore, an efficient sharing mechanism is required.

In this paper, a floating point divider performing quotient conversion and rounding in parallel is proposed by analyzing the operational characteristics of floating point division. The proposed floating point divider does not require any additional execution time, nor does it need any high speed adder for the rounding step. Also it can execute quotient conversion, rounding, and normalization within only one cycle, and can share the addition/rounding hardware logics for a floating point multiplier presented in [11] by adding several hardware logics. The additive hardware does not affect the execution time of the floating point multiplier and can be implemented with very simple hardware logics.

In [9], quotient conversion and rounding can be performed in parallel, and the addition/rounding hardware logics are shared with a floating point multiplier presented in [10]. However, it requires a more complex processing algorithm. This in turn requires additional hardware components which increase the length of the critical path on the pipeline. Such increase causes one more unit of pipeline delay in their approach and requires more hardware components than our approach.

The rest of this paper is organized as follows. Section 2 presents a brief overview of the IEEE rounding methods and the integer SRT division method. We also illustrate how the proposed floating point divider can share an addition/rounding unit of the floating point multiplier. Section 3 suggests a hardware model which can execute rounding and quotient conversion in parallel and its implementation with respect to IEEE four rounding modes. In Section 4, conclusion is given.

2 Backgrounds and Basic Equations

In this section, the IEEE rounding methods and the SRT division algorithm are discussed. An addition and rounding circuit for a floating point multiplier is also illustrated to be shared with the proposed floating point divider.

2.1 The IEEE Rounding Modes

The IEEE standard 754 stipulates four rounding modes; they are round-to-nearest, round-to-zero, round-to-positive-infinity, and round-to-negative-infinity. These four rounding modes can be classified mainly into round-to-nearest, round-to-zero, and round-to-infinity, because round-to-positive-infinity and round-to-negative-infinity can be divided into round-to-zero and round-to-infinity according to the sign of a number.

For the sake of the IEEE rounding, two additional bits, R and Sy, are required. R is the MSB among the less significant bits other than LSB. Sy is the ORed results of all the less significant bits except with R. The following three algorithms are the results of the rounding operation with LSB, R, and Sy when the MSB is zero, which is a normalized case. “return 0” means truncation, and “return 1” indicates increment as the result of any rounding operation.

Algorithm 1 : $Round_{nearest}(LSB, R, Sy)$

```
if ($R$=0) return 0
else if ($Sy$=1) return 1
     else if ($LSB$=0) return 0
          else return 1
```

Algorithm 2 : $Round_{zero}(LSB, R, Sy)$

```
return 0
```

Algorithm 3 : $Round_{infinity}(LSB, R, Sy)$

```
if (($R$=1) or ($Sy$=1)) return 1
else return 0
```

Assume that two input significands, *divisor* d and *dividend* x, have n bits each. To simplify the notation, the binary point is to be located between the LSB and the R bit positions. Then, the R and the Sy bit positions become the fraction portion. The significand bits above them, which are the most significant $(n + 1)$ bits, are the integer portion. The integer portion is represented with subscript I and the fractional portion with subscript T. Figure 1 shows that most significant $(n+1)$ bits of H are the integer portion H_I, while R and Sy in H are the fractional portion H_T.

For the rest of this paper, ‘$\wedge$’ denotes the boolean AND, ‘$\vee$’ denotes the boolean OR, and ‘$\oplus$’ denotes the boolean exclusive-OR. X denotes the “don’t care” condition. If any overflow is generated from the result of Z operation then $overflow(Z)$ returns 1, otherwise it returns 0. C_k^{in} denotes the value of the carry signal from the $(k-1)$-th bit position into the k-th bit position.

$$H = \overbrace{h_n h_{n-1} h_{n-2} \cdots h_1 h_0}^{H_I} . \overbrace{RSy}^{H_T}$$

Fig. 1. The definitions of H_I and H_T.

2.2 SRT Recurrence

Division operation can be defined by the following equation [6]:

$$x = q \times d + rem,$$

where $|rem| < |d| \times ulp$. The dividend x and the divisor d are the input operands. The quotient q and the remainder rem are the results of the division operation. The unit in the last position, denoted by ulp, defines the precision of the quotient, where $ulp = r^{-n}$ for n-digit and radix-r fractional results.

The following recurrence is used at each iteration:

$$rP_0 = x$$
$$P_{j+1} = rP_j - dq_{j+1},$$

where q_{j+1} is the $(j+1)$-th quotient digit, numbered from the highest to the lowest order, and P_j is the partial remainder at iteration j. In order for the next partial remainder P_{j+1} to be bounded, the value of the quotient-digit is chosen as follow:

$$|P_{j+1}| \leq d.$$

The final quotient is the weighted sum of all of the quotient-digits selected through the iteration such that

$$Q_{final} = \sum_{j=1}^{n} q_j \times r^{-j}.$$

In general, to speed-up the recurrence, the partial remainder and the quotient are represented as in the redundant forms. That is, the partial remainder consists of the carry part and the sum part, and the quotient consists of the positive part and the negative part.

2.3 Calculating the Final Quotient, *R*, and *Sy*

Suppose that the quotient obtained after the l-th iteration is denoted by Q_l and the partial remainder is denoted by P_l. Then because P_l should be positive, the restoration step for P_l is required to produce the final quotient if P_l is negative. Therefore, considering the restoration step, Q_l and P_l can be converted as follows.

$$\tilde{Q}_l = Q_l - restore_l$$
$$\tilde{P}_l = P_l + restore_l \cdot d$$
$$restore_l = \begin{cases} 1 & \text{if } P_l < 0 \\ 0 & \text{otherwise.} \end{cases}$$

After the completion of SRT recurrence, the quotient and the remainder values are generated in the redundant binary representations. The quotient value from the result of SRT recurrence consists of the positive part Q^P and the negative part Q^N. Then Q^P and Q^N can be shown as follows.

$$\begin{aligned} Q^P &= p_n \cdots p_0 . p_{-1} \\ Q^N &= n_n \cdots n_0 . n_{-1}, \end{aligned}$$

where Q^N is represented in the one's complement form.

The remainder value generated after the completion of SRT recurrence consists of the carry part Rem_C and the sum part Rem_S. Both Rem_C and Rem_S are represented in the two's complement form and have length of $(n+1)$ bits each. Rem_C and Rem_S can be now represented as follows.

$$\begin{aligned} Rem_S &= s_n \cdots s_1 s_0 \\ Rem_C &= c_n \cdots c_1 c_0 . \end{aligned}$$

The integer portions of Q^P and Q^N are denoted as Q_I^P and Q_I^N, respectively. Q_I^P and Q_I^N can be then defined as follows.

$$\begin{aligned} Q_I^P &= p_n \cdots p_0 \\ Q_I^N &= n_n \cdots n_0 . \end{aligned}$$

Then, because Q^N is represented in the one's complement form, when the values in the redundant representation converted into the conventional binary representation, '1' should be added to the position of n_{-1}. Also, if the remainder is negative, '1' should be borrowed from the quotient to restore the final remainder. Therefore, H which is the conventional binary representation of the final quotient including R and Sy can be calculated as follows.

$$\begin{aligned} H &= h_n h_{n-1} \cdots h_0 . RSy \\ &= (p_n \cdots p_0) + (n_n \cdots n_0) + 0.p_{-1} + 0.n_{-1} + \overline{restore_{-1}} + 0.0Sy, \end{aligned} \qquad (1)$$

where $restore_{-1}$ is the restoration signal to the (-1)-th position of the quotient and is identical to the sign value of the result of $(Rem_S + Rem_C)$. $Sy = 0$, if the result sum bits of $(Rem_S + Rem_C)$ are all zeros, otherwise $Sy = 1$.

Then, (1) can be converted as follows.

$$\begin{aligned} H &= Q_I^P + Q_I^N + 0.p_{-1} + 0.n_{-1} + 0.\overline{restore_{-1}} + 0.0Sy \\ &= Q_I^P + Q_I^N + C_0^{in} + 0.RSy \\ R &= p_{-1} \oplus n_{-1} \oplus \overline{restore_{-1}} \\ C_0^{in} &= overflow(p_{-1} + n_{-1} + \overline{restore_{-1}}), \end{aligned}$$

where R is the round bit and C_0^{in} is the value of the carry signal from the (-1)-th bit position to the 0-th bit position. Hence, H_I, the integer portion of H, can be represented as follows.

$$H_I = Q_I^P + Q_I^N + C_0^{in}. \qquad (2)$$

2.4 The Addition/Rounding Unit for a Floating-Point Multiplier

In general, processing floating point multiplication consists of multiplication, addition, rounding, and normalization steps. A floating point multiplier in [11, 12,13] can execute the addition/rounding operation within only one pipeline stage and hence simplify the hardware design. The hardware model presented in [11] is shown in Figure 2. The proposed floating point divider can share the addition/rounding unit for a floating point multiplier to process the quotient conversion, the rounding, and the normalization steps.

Sy which is calculated at the previous pipeline stage in parallel with a wallace tree is included in the input factors of the *predictor* logic in [11]. Because Sy can be generated after SRT recurrence in floating point division, Sy must not be included among the input elements for the *predictor* logic to perform addition and rounding in parallel. This is considered in the proposed floating point divider which is given in next section.

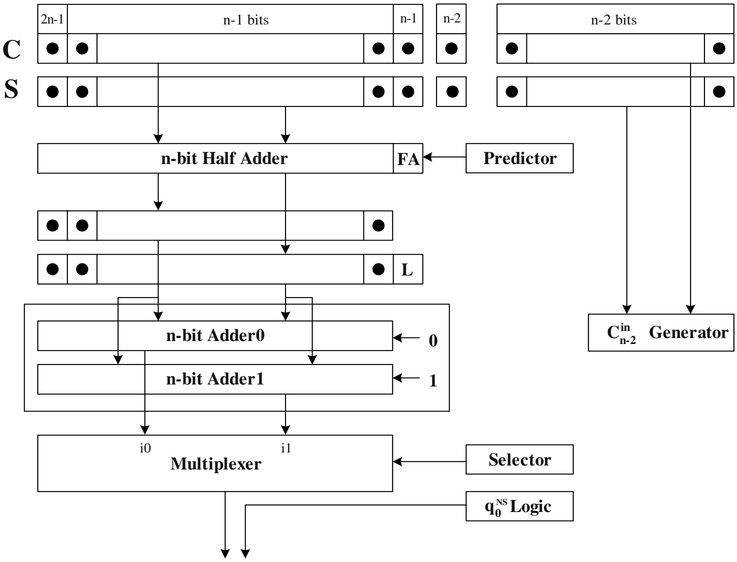


Fig. 2. The hardware model of the addition/rounding stage for a floating point multiplier.

3 The Proposed Floating-Point Divider

In this section, the operational characteristics in performing rounding and quotient conversion in parallel are illustrated. Based on these characteristics, a hardware model is presented. Finally, the proposed floating point divider is compared with respect to cycle time.

3.1 Analysis for the Quotient Conversion and Rounding Steps

Depending on the MSB of H, the shifting operation for normalization is performed. If $h_n = 1$ then no bit shifting for normalization is required, otherwise one bit shifting to the left should be performed in the normalization stage. The former case is denoted as NS (no shift) and the latter case is denoted as LS (left shift).

Normalization should be accounted for two different cases, i.e., the LS and the NS cases. In the LS case, the result value of H after normalization is denoted as O^{LS}. For the NS case, the result value of H after normalization is denoted as O^{NS}. Suppose that O_I^{LS}, O_R^{LS}, O_{Sy}^{LS} are the integer portion of O^{LS}, the R bit value, and the Sy bit value in the case of LS, respectively. Then they can be represented as follows.

$$\begin{aligned} O_I^{LS} &= h_{n-1}h_{n-2}\cdots h_0 \\ O_R^{LS} &= R \\ O_{Sy}^{LS} &= Sy. \end{aligned} \tag{3}$$

O_I^{NS}, O_R^{NS}, and O_{Sy}^{NS} are defined similarly for the NS case as follows:

$$\begin{aligned} O_I^{NS} &= h_n h_{n-1}\cdots h_1 \\ O_R^{NS} &= h_0 \\ O_{Sy}^{NS} &= R \vee Sy. \end{aligned} \tag{4}$$

Suppose that Q is the result value after the rounding step which is performed prior to the normalization stage. Then in the LS case, Q is represented by Q^{LS}. Thus, Q^{LS} can be written as follows according to (2) and (3):

$$\begin{aligned} Q^{LS} &= O_I^{LS} + Round_{mode}(h_0, R, Sy) \\ &= (h_{n-1}h_{n-2}\cdots h_0) + Round_{mode}(h_0, R, Sy) \\ &= Q_I^P + Q_I^N + C_0^{in} + Round_{mode}(h_0, R, Sy). \end{aligned} \tag{5}$$

Also, for the NS case, Q is represented by Q^{NS}. Thus, Q^{NS} can be obtained as follows according to (2) and (4).

$$\begin{aligned} Q^{NS} &= (O_I^{NS} + Round_{mode}(h_1, h_0, (R \vee Sy))) \times 2 \\ &= (h_n \cdots h_1 h_0) + 2 \times Round_{mode}(h_1, h_0, (R \vee Sy)) \\ &= Q_I^P + Q_I^N + C_0^{in} + 2 \times Round_{mode}(h_1, h_0, (R \vee Sy)). \end{aligned} \tag{6}$$

3.2 The Proposed Hardware Model for Performing IEEE Rounding and Quotient Conversion in Parallel

A hardware model capable of performing rounding and quotient conversion in parallel is designed as shown in Figure 3. The proposed hardware model can be implemented by adding some hardware to the hardware model in Figure 2.

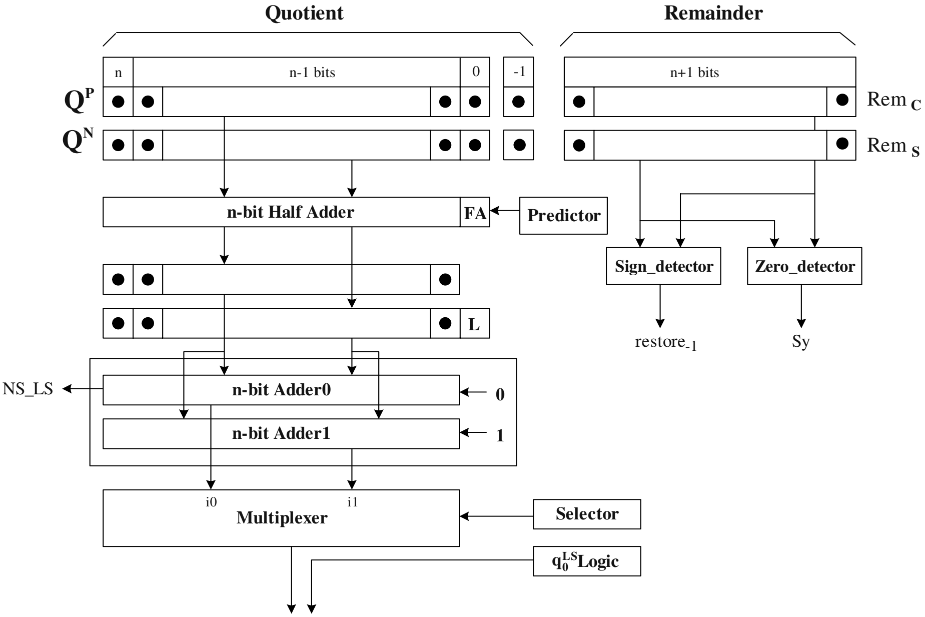


Fig. 3. The proposed hardware model for performing IEEE rounding and quotient conversion in parallel.

In *sign_detector*, the result value of $restore_{-1}$ in the (1) is calculated. The *zero_detector* logic determines whether the remainder is zero. If the result of *zero_detector* is zero, $Sy = 0$, otherwise $Sy = 1$. These logics can be implemented by adding additional logics to the C^{in}_{n-2} *generator* in Figure 2.

When Q^P_I and Q^N_I are added by the n bit HA and the one bit FA, the *predictor* bit is provided to the FA. Then the n bit carry and the $(n+1)$ bit sum are generated. Here, the LSB of the sum is represented by L as shown in Figure 3. The n bit carry and the most significant n bit sum are added by a single carry select adder which is drawn as a dotted box in Figure 3. The *selector* selects one of the result values after executing addition and rounding from the two inputs $i0$ and $i1$. If $selector = 0$, then $i0$ is selected, otherwise $i1$ is selected as the output value of the multiplexer. The input values of $i0$ and $i1$ can be represented as follows.

$$\begin{aligned} i0 &= Q^P_I + Q^N_I + predictor \\ i1 &= Q^P_I + Q^N_I + 2 + predictor. \end{aligned} \tag{7}$$

In Figure 3, the multiplexer output may be either $(Q^P_I + Q^N_I + predictor)$ or $(Q^P_I + Q^N_I + predictor + 2)$, depending on the value of *selector*. According to (5) and (6), one of four possible cases, i.e., $(Q^P_I + Q^N_I)$, $(Q^P_I + Q^N_I + 1)$, $(Q^P_I + Q^N_I + 2)$, and $(Q^P_I + Q^N_I + 3)$ needs to be generated to perform rounding and quotient conversion in parallel. Therefore, if *predictor* and *selector* are properly selected, then the result value Q after performing addition and rounding in parallel can be generated. Note that Q is defined in Section 3.1.

To configure *predictor*, the following two factors should be considered. First, the input signals of *predictor* must be generated before any addition operation performed by the carry select adder. Second, the delay of the *selector* logic which is finally configured after the determination of the *predictor* logic should be negligible. Thus, *predictor* must be selected very carefully.

The input value of $i0$ in the multiplexer is denoted by $E = e_n e_{n-1} \cdots e_1$. Because the LSB position of E corresponds to the first bit positions of Q^P and Q^N, the integer value of E is $E_I = E \times 2$. Thus, the $(n+1)$ bit integer field can be denoted as E_I^* and $E_I^* = E_I + L$. Hence, E_I^* can be represented as follows:

$$E_I^* = E_I + L = Q_I^P + Q_I^N + predictor = e_n \cdots e_1 L. \tag{8}$$

In the next subsections, the rounding position is analyzed, and *predictor* and *selector* are determined according to all the three rounding modes.

3.3 The Round-to-Nearest Mode

In the round-to-nearest mode and the LS case, one of three possible cases, i.e., $(Q_I^P + Q_I^N)$, $(Q_I^P + Q_I^N + 1)$, and $(Q_I^P + Q_I^N + 2)$ needs to be generated according to (5) to perform rounding and quotient conversion in parallel.

In the NS case, for increment as the result of $Round_{Nearest}(h_1, h_0, (R \vee Sy))$, h_0 should be '1' according to Algorithm 1. If the result of rounding is increment and the NS case, '1' should be added to the position of h_1. But because h_0 should be '1' for increment as a result of rounding in the NS case, adding '1' to the position of h_1 has the same most significant n bit result when '1' is added to the position of h_0. Therefore, one of the three possible cases, i.e., $(Q_I^P + Q_I^N)$, $(Q_I^P + Q_I^N + 1)$, and $(Q_I^P + Q_I^N + 2)$, is required to be generated also in the NS case.

According to (7), when *predictor* is selected to '0', $(Q_I^P + Q_I^N)$ and $(Q_I^P + Q_I^N + 2)$ are generated. Because the most significant n bits of either $(Q_I^P + Q_I^N)$ or $(Q_I^P + Q_I^N + 2)$ are identical to the most significant n bits of $(Q_I^P + Q_I^N + 1)$, $(Q_I^P + Q_I^N + 1)$ can be generated. However, *predictor* is selected as follows to simplify the *selector* logic.

$$predictor = p_{-1} \wedge n_{-1}. \tag{9}$$

Then, (2) can be converted as follows.

$$\begin{aligned} H &= Q_I^P + Q_I^N + 0.p_{-1} + 0.n_{-1} + 0.\overline{restore_{-1}} + 0.0Sy \\ &= Q_I^P + Q_I^N + overflow(p_{-1} + n_{-1}) + 0.(p_{-1} \oplus n_{-1}) + \\ &\quad\; 0.\overline{restore_{-1}} + 0.0Sy \\ &= Q_I^P + Q_I^N + predictor + 0.(p_{-1} \oplus n_{-1}) + 0.\overline{restore_{-1}} + 0.0Sy \\ &= E_I + L + C_0^{in} + 0.R + 0.0Sy \\ &= E_I + C_1^{in} \times 2 + h_0 + 0.R + 0.0Sy \end{aligned} \tag{10}$$

$$\begin{aligned} R &= (p_{-1} \oplus n_{-1}) \oplus \overline{restore_{-1}} \\ C_0^{in} &= (p_{-1} \oplus n_{-1}) \wedge \overline{restore_{-1}} \\ h_0 &= L \oplus C_0^{in} \\ C_1^{in} &= L \wedge C_0^{in}. \end{aligned}$$

In the LS case, Q^{LS} can be expressed as follows according to (5) and (10).

$$Q^{LS} = E_I + C_1^{in} \times 2 + h_0 + Round_{nearest}(h_0, R, Sy). \tag{11}$$

Then, $selector$ and q_0^{LS} can be produced as follows.

$$\begin{aligned} selector &= C_1^{in} \vee (h_0 \wedge Round_{nearest}(h_0, R, Sy)) \\ q_0^{LS} &= h_0 \oplus Round_{nearest}(h_0, R, Sy). \end{aligned}$$

In the NS case, Q^{NS} is as follows according to (6) and (10).

$$Q^{NS} = E_I + 2 \times C_1^{in} + h_0 + 2 \times Round_{nearest}(h_1, h_0, R \vee Sy). \tag{12}$$

Then, $selector$ can be produced as follows.

$$selector = C_1^{in} \vee Round_{nearest}(h_1, h_0, R \vee Sy).$$

3.4 The Round-to-Zero Mode

It is considered that the *predictor* of the round-to-zero mode is identical to that of the round-to-nearest mode. Because $Round_{Zero}(X, X, X) = 0$ for both the LS and the NS cases, both *selector* and q_0^{LS} can be obtained by replacing both $Round_{Nearest}(h_0, R, Sy)$ and $Round_{Nearest}(h_1, h_0, R \vee Sy)$ of (11) and (12) with zeros. Therefore, *selector* and q_0^{LS} can be written as follows:

$$\begin{aligned} selector &= C_1^{in} \\ q_0^{LS} &= h_0. \end{aligned}$$

3.5 The Round-to-Infinity Mode

For a specific case such as $L = p_{-1} \oplus n_{-1} = C_0^{in} = Sy = 1$, if the *predictor* of the round-to-nearest mode is used, then $C_1^{in} = 1$, $h_0 = 0$, $R = 0$, and $Sy = 1$. Thus the rounding result in the round-to-nearest mode can be obtained by truncation according to Algorithm 1. However, because the value of Sy is equal to '1', the rounding result in the round-to-infinity mode can be obtained by increment according to Algorithm 3. Therefore, in the NS case, the case of $C_1^{in} = 1$ and $Round_{infinity} = 1$ can be occurred. Eventually, in the round-to-infinity mode, the *predictor* of the round-to-nearest mode cannot be used. Thus, the *predictor* is given as follows.

$$predictor = p_{-1} \vee n_{-1}. \tag{13}$$

Table 1. The result values of C_0^{in}, R, *predictor* according to the values of the p_{-1}, n_{-1}, $\overline{restore_{-1}}$.

p_{-1}	n_{-1}	$\overline{restore_{-1}}$	$p_{-1} \oplus n_{-1}$	C_0^{in}	R	$predictor$
0	0	0	0	0	0	0
0	1	0	1	0	1	1
1	0	0	1	0	1	1
1	1	0	0	1	0	1
0	0	1	0	0	1	0
0	1	1	1	1	0	1
1	0	1	1	1	0	1
1	1	1	0	1	1	1

In Table 1, C_0^{in}, R, and *predictor* are illustrated as the values of p_{-1}, n_{-1}, and $\overline{restore_{-1}}$. The following two cases can be generated according to the values of $p_{-1} \oplus n_{-1}$ and $\overline{restore_{-1}}$. First, $\overline{restore_{-1}} = 0$ and $p_{-1} \oplus n_{-1} = 1$. In this case, C_0^{in} which represents the carry value to the position of h_0 is 0, both R and *predictor* are 1's. Second, the value of C_0^{in} has an identical value with *predictor* for all the cases except the first case.

In the first case, the result of rounding is increment because $R = 1$ according to Algorithm 3, $predictor = 1$ and $C_0^{in} = 0$. Then, Q^{LS} can be given as follows according to (5) and (8).

$$\begin{aligned} Q^{LS} &= Q_I^P + Q_I^N + C_0^{in} + Round_{infinity}(h_0, R, Sy) \\ &= Q_I^P + Q_I^N + 1 = Q_I^P + Q_I^N + predictor \\ &= E_I + L. \end{aligned}$$

Then, *selector* and q_0^{LS} can be produced as follows in this case.

$$\begin{aligned} selector &= 0 \\ q_0^{LS} &= L. \end{aligned} \tag{14}$$

In the NS case, Q^{NS} can be written as follows according to (6) and (8).

$$\begin{aligned} Q^{NS} &= Q_I^P + Q_I^N + C_0^{in} + 2 \times Round_{infinity}(h_0, R, Sy) \\ &= Q_I^P + Q_I^N + 2 = Q_I^P + Q_I^N + predictor + 1 \\ &= E_I + L + 1. \end{aligned}$$

Hence, *selector* is as follows in this case.

$$selector = L.$$

In the second case, the value of C_0^{in} is identical to *predictor*. Then, Q^{LS} can be produced as follows.

$$\begin{aligned} Q^{LS} &= Q_I^P + Q_I^N + C_0^{in} + Round_{infinity}(h_0, R, Sy) \\ &= Q_I^P + Q_I^N + predictor + Round_{infinity}(h_0, R, Sy) \\ &= E_I + L + Round_{infinity}(h_0, R, Sy). \end{aligned}$$

Then, in this case, *selector* and q_0^{LS} can be determined as follows.

$$\begin{aligned} selector &= L \wedge Round_{infinity}(h_0, R, Sy) \\ q_0^{LS} &= L \oplus Round_{infinity}(h_0, R, Sy). \end{aligned}$$

Thus, Q^{NS} is also obtained as follows.

$$\begin{aligned} Q^{NS} &= Q_I^P + Q_I^N + C_0^{in} + 2 \times Round_{infinity}(h_1, h_0, R \vee Sy) \\ &= Q_I^P + Q_I^N + predictor + 2 \times Round_{infinity}(h_1, h_0, R \vee Sy) \\ &= E_I + L + 2 \times Round_{infinity}(h_1, h_0, R \vee Sy). \end{aligned}$$

Hence, *selector* is represented as follows in this case.

$$selector = Round_{infinity}(h_1, h_0, R \vee Sy).$$

3.6 Critical Path Analysis

There are two dataflows in Figure 4. The critical path latency of the left hand side flow, denoted as L_{CP}^{div}, is ($predictor + FA +$ carry select adder $+ selector +$ multiplexer). The right hand side flow, denoted as R_{CP}^{div}, is ($zero\_detector + selector +$ multiplexer). In L_{CP}^{div} and R_{CP}^{div}, q_0^{LS} is ignored because both *selector* and q_0^{LS} can be performed simultaneously and the delay characteristic of *selector* is more complex than that of q_0^{LS}. As aforementioned in Section 2.3, the result values of $restore_{-1}$ and Sy can be calculated with the result of $Rem_C + Rem_S$. Because the latency of *zero_detector* is either equal to or longer than that of *sign_detector*, *sign_detector* is not also included in R_{CP}^{div}.

If we compare L_{CP}^{div} with R_{CP}^{div}, it seems that the carry select adder and *zero_detector* reveal similar delay characteristic, because both can be implemented with a high speed adder. Thus, the critical path of the hardware model in Figure 4 will be L_{CP}^{div}.

L_{CP}^{div} is almost similar to the critical path L_{CP}^{mul} of the hardware model in Figure 2. Only the delay characteristics of *predictor* and *selector* are somewhat different. The logic delay of *predictor* on L_{CP}^{div} is one gate delay for each rounding mode, otherwise that of L_{CP}^{mul} is two gate delays in the case of the round-to-infinity mode. For the *selector* of L_{CP}^{div}, an additional exclusive-OR gate is required due to identify the two cases in Table 1, as shown in Section 3.5. The additive gate delay of L_{CP}^{div} amount to the gate delays of exclusive-OR minus one gate delay. This additive delay is so small that it may not affect the overall pipeline latency.

3.7 Comparison with On-the-Fly Rounding

In [14], on-the-fly rounding was suggested to avoid a carry-propagation addition in rounding operation, by combining the rounding process with the on-the-fly conversion of the quotient digits from redundant to conventional binary form. The on-the-fly rounding requires one cycle for the rounding operation because the

Table 2. The execution cycles for double precision according to radix-r.

Processor	cycles	radix
PowerPC604e [3]	31	4
PA-RISC 8000 [15]	31	4
Pentium [7]	33	4
UltraSPARC [4]	22	8
R10000 [5]	19	16
Proposed floating point divider	30	4
Proposed floating point divider	21	8
Proposed floating point divider	15	16

rounded quotient is selected after the sign bit detection. This one cycle latency is equal to the latency for the rounding operation of the proposed architecture. But, the on-the-fly rounding requires four shift registers with somewhat complex parallel load operations.

On the other hand, the proposed architecture can share the addition/rounding hardware logics for a floating point multiplier presented in [14]. Thus, the proposed architecture seems to require less hardware than the on-the-fly rounding. Moreover, the proposed architecture achieves low-power consumption over the on-the-fly rounding, because the parallel loading operations for four registers could be generated at each iteration in case of on-the-fly rounding, while only one rounding operation is required in the proposed architecture.

3.8 Comparison with Other Microprocessors

To complete double precision floating point division, the SRT recurrences on the radix-4 case, on the radix-8 case, and on the radix-16 case take 29 cycles, 20 cycles, and 14 cycles, respectively. Because the proposed floating point divider can perform quotient conversion and rounding within only one cycle, to complete the floating point division operation, 30 cycles should be taken in the radix-4 case, 21 cycles for the radix-8 case, and 15 cycles for the radix-16 case, respectively. As shown in Table 2 in the radix-4 case, PowerPC604e [3] takes 31 cycles, PA-RISC 8000 [15] takes 31 cycles, and Pentium [7] takes 33 cycles to complete the floating point division operation. In the radix-8 case, UltraSPARC [4] takes 22 cycles. Also, R10000 [5] takes 19 cycles in the radix-16 case.

4 Conclusion

In this paper, a floating point divider which is capable of performing the IEEE rounding and addition in parallel is proposed. Its hardware model is provided and evaluated with the proofs for correctness of the model. The performance improvement and cost effectiveness design for floating point division can be achieved by this approach.

References

1. M. Kameyama, Y. Kato, H. Fujimoto, H. Negishi, Y. Kodama, Y. Inoue, and H. Kawai. 3D graphics LSI core for mobile phone "Z3D". In Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics hardware, pages 60–67, 2003.
2. S. Oberman, G. Favor, and F. Weber. AMD 3DNow! technology: architecture and implementations. IEEE Micro, 19(2):37–48, April 1999.
3. S. P. Song, M. Denman, and J. Chang. The powerPC604 RISC microprocessor. IEEE Micro, 14(5):8–17, Oct. 1994.
4. M. Tremblay and J. M. O'Connor. Ultra SPARC I : A four-issue processor supporting multimedia. IEEE Micro, 16(2):42–50, April 1996.
5. K. C. Yeager. The MIPS R10000 superscalar microprocessor. IEEE Micro, 16(2):28–40, April 1996.
6. M. D. Ercegovac and T. Lang. Division and square root: digit recurrence algorithms and implementations. (Kluwer Academic Publishers, 1994).
7. D. Alpert and D. Avnon. Architecture of the Pentium microprocessor. IEEE Micro, 13(3):11–21, June 1993.
8. C. H. Jung, W. C. Park, T. D. Han, S. B. Yang, and M. K. Lee. An effective out-of-order execution control scheme for an embedded floating point coprocessor. Microprocessors and Microsystems, 27:171–180, April 2003.
9. J. A. Prabhu and G. B. Zyner. 167 MHz Radix-8 divide and square root using overlapped radix-2 stages. In Proceedings of the 12th IEEE Symposium on Computer Arithmetic, pages 155–162, July 1995.
10. J. Arjun Prabhu and Gregory B. Zyner. 167 MHZ radix–4 floating point multiplier. In Proceedings of the 12th IEEE Symposium on Computer Arithmetic, pages 149–154, July 1995.
11. W. C. Park, T. D. Han, and S. D. Kim. Floating point multiplier performing IEEE rounding and addition in parallel. Journal of Systems Architecture, 45:1195–1207, July 1999.
12. G. Even and P. M. Seidel. A comparison of three rounding algorithms for IEEE floating-point multiplication. IEEE Transactions on Computers, 49(7):638–650, July 2000.
13. M. R. Santoro, G. Bewick, and M. A. Horowitz. Rounding algorithms for IEEE multiplier. In Proceedings of the 9th IEEE Symposium on Computer Arithmetic, pages 176-183, 1989.
14. M. D. Ercegovac and T. Lang. On-the-fly Rounding. IEEE Transactions on Computers, 41(12):1497–1503, Dec. 1992.
15. P. Solderquist and M. Leeser. Division and square root: choosing the right implementation. IEEE Micro, 17(4):56–66, Aug. 1997.

Efficient Buffer Allocation for Asynchronous Linear Pipelines by Design Space Localization

Jeong-Gun Lee[1], Euiseok Kim[2], Jeong-A Lee[3], and Eunok Paek[4]

[1] Department of Information and Communications,
Gwangju Institute of Science and Technology, Republic of Korea
eulia@gist.ac.kr

[2] Samsung Advanced Institute of Technology, Republic of Korea
euiseok2003.kim@samsung.com

[3] Department of Computer Engineering, Chosun University, Republic of Korea
jalee@chosun.ac.kr

[4] Department of Mechanical and Information Engineering,
University of Seoul, Republic of Korea
paek@uos.ac.kr

Abstract. Asynchronous circuit design is very attractive as a high performance design method since it can achieve average-case delay. However, it is hard to make use of such an advantage in a pipelined architecture due to the blocking/starvation effects between stages. In most of current solutions, buffers are allocated to reduce the blocking/starvation effects but it is difficult to find a distribution of buffers over an asynchronous linear pipeline(ALP) that is optimal in terms of 'time*area' cost.

In this paper, we show that the design space of the buffer allocation on an ALP is non-convex by introducing a term, called additional cycle time reduction (ACTR) that can separate the effect of a simultaneous buffer insertion from an individual buffer insertion. Furthermore, we propose a hybrid algorithm such that hill-climbing search is first performed during the early stage of buffer allocation while more sophisticated simulated annealing is applied for the later stage. Such a hybrid approach makes use of the characteristics of buffer allocation design space. Experiments and comparison with conventional methods based on simulated annealing are presented to show the efficiency of the proposed algorithm.

1 Introduction

Recently, asynchronous designs have been reconsidered as a high performance and low power system design method by incorporating advanced circuit design techniques. In particular, the benefit of average case performance, characterized by data dependent operations, is very attractive in designing high-performance systems. In order to achieve the expected average case performance of asynchronous circuits, a lot of efforts have been made considering various aspects of the design problem [6,7,8]. Difficulties arise, however, when those circuits and blocks are combined in a pipeline, because the pipeline may not show the average case performance or may even show near worst-case performance due to

P.-C. Yew and J. Xue (Eds.): ACSAC 2004, LNCS 3189, pp. 582–595, 2004.

blocking and starvation effects. Thus, it is strongly required to provide a way to achieve a near average case performance for asynchronous pipelines.

Asynchronous FIFO buffers have been considered as one of the best ways to achieve the average case performance in pipelines [5,6,9] as they help avoid blocking and starvation effects between pipeline stages. However it is difficult to answer the question, "How should the buffers be distributed in the circuit to ensure optimum time performance and time*area performance?" [9]. The difficulty comes from a non-convex characteristic and an enormous design space of the buffer allocation problem.

In this paper we propose an efficient 'performance*area' optimization method for an asynchronous linear pipeline (ALP) where the processing delay in each stage varies. In particular, we investigate the characteristics of buffer design space of ALPs and then suggest the localized space where the optima exist thus avoid exploring the entire buffer design space. Finally we propose an efficient buffer allocation algorithm exploring only a part of the design space based on the localization.

The paper is organized as follows. In Section 2, the previous work on performance analysis for an ALP is presented. An ALP model and its performance characteristics are described briefly in Section 3. In Section 4, we analyze optimums on the buffer design space of an ALP and present how to localize the search space. Section 5 proposes an efficient buffer allocation method based on the results presented in Section 4. Experimental results of the proposed algorithm are presented in Section 6. Finally, we conclude this paper in Section 7.

2 Related Work

Performance analysis and characterization have been done for asynchronous pipelined circuits in order to improve the speed of those pipelines through analytic approaches. In [2], the performance characterization of self-timed rings was performed and three types of performance region were classified as "data-limited", "bubble-limited", and "handshake-limited". For each classified performance region, the optimal number of tokens or stages was calculated in order to obtain best performance. However, only a fixed delay was used as a processing delay at each stage. In [3], asymptotic performance characteristics of an asynchronous pipeline were derived with probabilistic processing delay distributions. In [5], a coefficient variation for variable computation delay was used as an important factor affecting the system throughput. Using the coefficient variation, approximate performance formulas were derived through the empirical analysis. Then, the formulas were used to find the size of buffers required to achieve the specific rate of average case performance. In this work, it is assumed that all stages have identical and two-valued random delays only. Due to these restrictive and unrealistic assumptions, the formulas cannot be used effectively in real world designs with heterogeneous and complex logic stages.

Most recently, the limit for the average case performance was theoretically presented with diffusion equations [9] and percolation theory [11] independently.

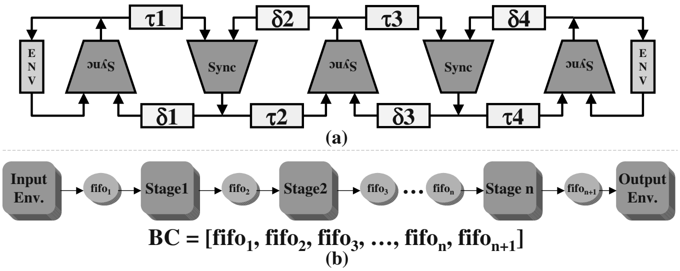


Fig. 1. (a) An ALP, (b) A conceptual model of the ALP and a buffer configuration

3 Target Model and Problem Description

3.1 An ALP Model and Its Performance

Fig.1(a) shows a four-stages ALP where τ_i is a forward latency and δ_i is a backward latency. The Sync module in the figure represents a synchronization mechanism between stages. The simplest implementation of Sync is a C-element, which is popular in asynchronous circuit design [4].

Each stage in the ALP has variable processing delays, that is, τ_i varies depending on the data being processed in each stage. Fig.1(b) illustrates an abstract model of an ALP. The first and the last stages communicate with input/output environments respectively. A buffer can be inserted to a channel to decouple operations of the two neighboring stages. To describe a buffer distribution over channels, a non-negative integer vector BC, a buffer configuration, is used. The dimension of BC is the same as the number of channels and BC[i] represents the number of buffers allocated to the channel 'i'.

Performance Characteristics of an ALP: Two special performance features of an ALP are shown in Figs.2. Parameters τ_i and δ_i in an ALP are assumed to be random variables with a uniform distribution in the interval [2, 12]. The delay of sf Sync is set to 0.5 unit delay, respectively. The environment is assumed to respond to an ALP immediately.

The dashed line in Fig.2(a) shows the performance of an ALP when the number of stages in an ALP increases. As the number of stages increase, the probability of blocking/starvation between stages becomes higher and therefore average cycle time becomes worse. Buffers can be used to reduce the blocking/starvation effects and improve the utilization of a functional logic in each stage by temporarily storing data between stages.

The solid line in Fig.2(b) illustrates the performance of an ALP, when buffers are sequentially inserted one by one to one channel in an ALP. The cycle time and reduction rate decrease monotonically and eventually converges to a certain point. The *exponential decrease of efficacy* with the number of buffers has also been observed in the simulation studies by [5]. When buffers are sufficiently allocated to *all the channels* in an ALP, the cycle time approaches to the average case.

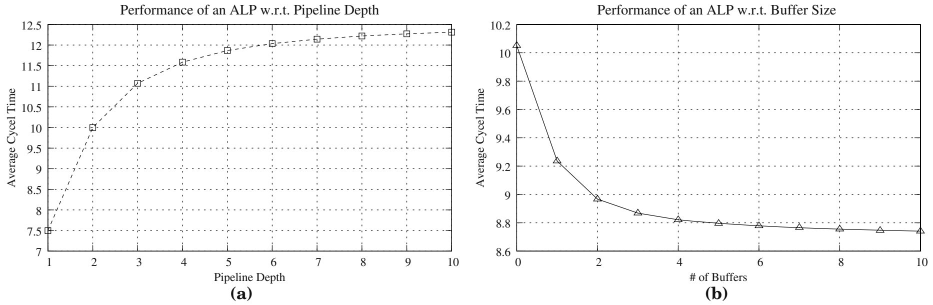


Fig. 2. Performance Impact of *Pipeline Depth* and *Buffer Size*

3.2 Terms and Problem Description

An ALP consisting of n stages has n+1 channels as shown in Fig.1(b). A buffer design space is composed of a set of possible buffer configurations. Assume that the number of available buffers is bn and the number of channels is cn. When i buffers are assigned to cn possible places, the number of i-combinations with repetition allowed from a set of cn channels is ${}_{i+|cn|-1}C_i$. Thus the size of a buffer design space is expressed by the following equation.

$$\Sigma_{i=0}^{bn}\ {}_{i+|cn|-1}C_i, \text{ where } C \text{ denotes a combination.}$$

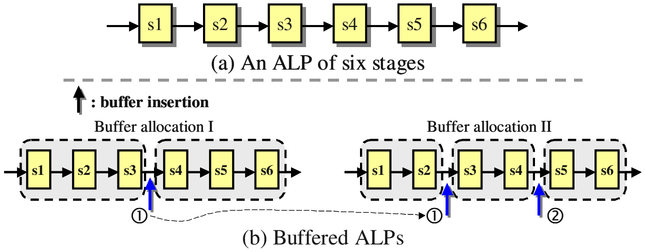
The equation implies that the size of buffer design space can increase significantly fast as the number of channels or the number of available buffers increases. For example, suppose that an ALP has 20 channels and 20 available buffers, and that the simulation consumes 0.01 second for each buffer configuration. Then the overall processing time required to determine an optimal buffer configuration is $[1.37847 * 10^{11}] * 0.01 = 1.37847 * 10^9$ seconds (about 43 years).

As a cost function, 'performance×area' is used. For the performance metric, average cycle time of an ALP under a given buffer allocation is evaluated. The area is defined by the sum of functional logic and buffer area. The cost function for a given buffer configuration, BC, is expressed by the following equation.

$$\begin{aligned} cost(\mathrm{BC}) &= performance \times area \\ &= cycle\_time(\mathrm{BC}) \times (function\_area + \sum_{\forall \mathrm{Channel}_i} \mathrm{BC}[\mathrm{Channel}_i] \times one\_buffer\_cell\_area) \end{aligned}$$

With this cost definition, the optimal buffer allocation problem is described as "finding a buffer configuration BC with the lowest cost"

A buffer configuration in a buffer design space can be considered as a state in a search-space search [1]. From a state, a set of buffer configurations can be derived by inserting/eliminating a buffer to/from a certain channel. The action of the buffer insertion or elimination is called a move. A set of states that can be derived from a state S by a single move is called a neighbor set or neighborhood



(a) An ALP of six stages

(b) Buffered ALPs

Fig. 3. Interaction between buffer allocations

of S. A gain of a move from BC to BC' is defined as the value of cost(BC) - cost(BC'). Thus, a positive gain of a move implies that cost is reduced by the move. The term, optimum is used to denote a local or a global optimum. When we need to distinguish between local and global optima, we use 'local' or 'global' explicitly. In this paper, a set of local optimums does not include global optimums. Finally, a convex design space is the space where only a single optimum exists. A non-convex space includes multiple optimums so a search can be trapped in a local optimum before reaching a global optimum in this space.

4 Buffer Allocation to an ALP

4.1 Local Optimums in a Buffer Design Space

Let us consider an ALP shown in Fig.3(a). Each stage is assumed to have a variable delay generation function that is identical across all the stages. It is easy to see that a local optimum can differ from a global optimum when we compare two simple cases of buffer allocation. One is allocating a single buffer and the other is simultaneously allocating two buffers. Let us assume that in the first case, "Buffer allocation I" shown in Fig.3(b) results in the biggest cost reduction. Also assume that in the second case, "Buffer allocation II" in Fig.3(b) generates the biggest cost reduction. Allocation of one buffer between stages s3 and s4, that enabled the biggest cost reduction in one-buffer allocation, prevents us from reaching the optimum in two-buffer allocation. However this phenomenon is unavoidable in a neighborhood-based search because it decides a search direction based only on neighbors reachable by a single move. Thus, this kind of a neighborhood-based search is easy to be trapped in a local optimum.

The best (optimum) position of one buffer changes when the other buffer is allocated additionally because *the cost reduction of one buffer allocation is affected by the other buffer allocation*. In order to quantify such an interaction between buffer allocations, we define a quantity called additional cycle time reduction (ACTR). Fig.4 shows a possible ACTR for the case of two-buffer allocation. ACTR shown in this figure is calculated by varying the channel positions of two buffers and is expressed in the following equation.

$$\mathrm{ACTR}(\{i,j\}) = \mathrm{CTR}(\{i,j\}) - \{\mathrm{CTR}(\{i\}) + \mathrm{CTR}(\{j\})\} \text{ — } (1)$$

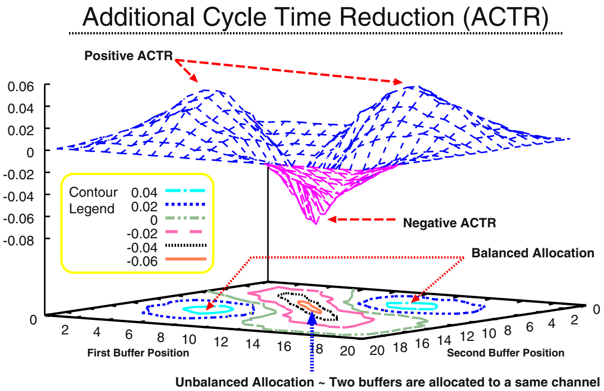


Fig. 4. Quantity of ACTR

The function CTR takes a multi-set[1] CP of channels as arguments and returns cycle time reduction achieved by allocating a buffer to each channel in CP (negative numbers can be used in the set CP to denote a buffer elimination). ACTR can be positive or negative depending on a buffer distribution over the channels, and it can be thought as an additional effect caused by an interaction between the two concurrent buffer insertions.

Particularly, a positive ACTR implies that *considering multiple moves simultaneously makes cycle time to be reduced further and results in a more cost reduction that is not foreseeable by any single move of a local search.* These kinds of interactions are the main causes of producing local optima in the buffer design space of an ALP since ACTR is the only way of generating local optima by invisible additional cost reduction. The following proposition addresses the relationship.

Proposition 1. *A buffer configuration* BC *is a local optimum iff the following two conditions are satisfied.*
1. The cost of BC *is lower than the costs of all of its neighbors.*
2. A positive ACTR is caused by a set of buffer insertions / eliminations that are simultaneously performed from BC *and such set of moves leads to another buffer configuration* BC' *whose cost is lower than that of* BC.

Notice that a local optimum is described in terms of ACTR. In the following, through observing the behavior of cycle time reductions and ACTR effects in buffer design space, the characteristic of buffer design space is investigated.

4.2 A Characteristic of a Buffer Design Space

When only a single channel is concerned, thanks to the monotonicity of cycle time change as shown in Fig.2(b), the cost also monotonically changes. Conse-

[1] A multi-set is a set that may have duplicates. For example, A = {1,1,2,2,2,3} is a multi-set and it is not the same as {1, 2, 3}.

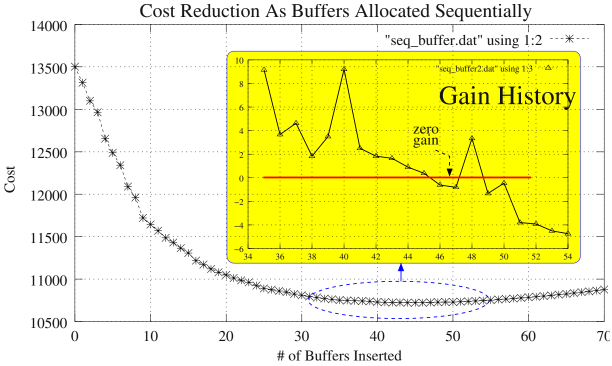


Fig. 5. A sequential buffer allocation and its corresponding cost reduction

quently, an optimal number of buffers at the channel can be found by checking the gradient on the cost surface.

When all the channels are concerned, however, finding a global optimum becomes computationally difficult due to the non-convexity of design space. Nevertheless, as shown in Fig.5, when the buffers are inserted sequentially to the channels from a zero buffer configuration (BC = $\mathbf{0}$) while making the biggest cost reduction in each step, an initial part of the design space looks like a convex space. The cost decreases monotonically up to a certain value due to the high efficacy of initial buffer insertions.

As more buffers are inserted to each channel, the efficiency shrinks and the cost reduction becomes limited, and ACTR effects begin to generate local optima by satisfying the first condition of Proposition 1. In other words, initial insertions to each channel show relatively higher tendency of being on a way to a global optimum than later insertions. In an optimally buffered ALPs, latest insertions to each channel are prone to move away from a global optimum. Therefore, latest allocations to each channel need to be rearranged to find a global optimum. Since cycle time reduction achieved by these recent allocations is too small to reduce cost, the quantity of an ACTR becomes small too.

The graph drawn in Fig.6 shows an experimental evidence for the aforementioned claim. In this graph, the 'Max(Avg) Variation Effect' shows maximum (average) quantity of cycle time reduction achieved by a single buffer insertion at each sequential insertion step. Similarly, 'Max(Avg) ACTR Effect' is defined as maximum (average) quantity of cycle time reduction additionally achieved by simultaneous multiple buffer insertions.

As buffers are inserted to ALPs, the difference between a variation effect and an ACTR effect becomes smaller as well as their absolute values. There exists a point where cycle time reduction by single buffer insertions is not enough to reduce the overall cost. In the vicinity of this point, ACTR may shape like hills that correspond to local optima. Note that the quantity of ACTR at this point is so small that taking a small uphill move can lead to a global optimum.

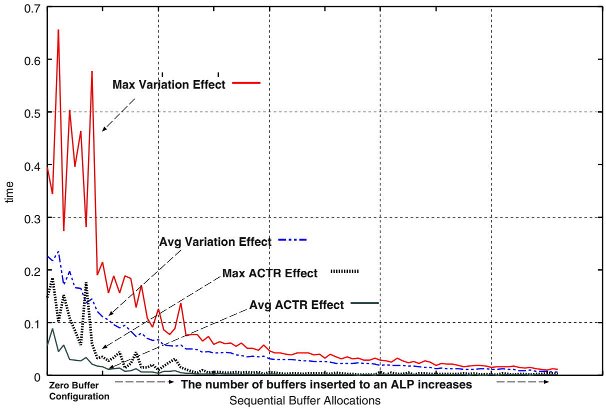


Fig. 6. A sequential buffer allocation and its corresponding ACTR

The equation (1) says that the cost reduction between two optima is determined by cycle time reduction CTR(CP), where CP is a set of moves that connects two optima. In the vicinity of the point where cycle time reduction saturated, the cycle time reduction of an individual move increases cost of the corresponding state and this causes uphill as a result. For CTR(CP) to be sufficient in order to reduce cost further, that is implying that a more optimized buffer configuration is obtained by accepting the set of moves, ACTR should be large enough to compensate the cost increases by the individual moves. Since the ACTR becomes small at the vicinity of the point, the cost increases by the individual moves is limited by the ACTR. In consequence, uphill height is also likely to be small. Finally, it implies that a search can escape from a local optimum with a small uphill potential barrier and this is substantiated by experiments in Section 6.

Fig.7 shows the simplified cost reduction graph where the part of non - monotonous cost surface is enlarged. In this figure, the horizontal axis represents the number of buffers inserted to a buffer configuration. The bowl shape of the cost line indicates an optimum since there is no move that can result in cost reduction. Since the cost reduction of moves can be described by a gain, the region in which optima appear can be detected by tracing the gain history presented as the shadowed graph embedded in Fig.5.

4.3 Localization of a Region

In the proposed buffer allocation algorithm, a buffer design space can be partitioned into two sorts of disjoint subspaces: (1) an Optimum Region (OPR) including optima and (2) a Non Optimum Region (NOPR) containing no optimum. OPR and NOPR are demarcated by a cost boundary as shown in Fig.8, where the horizontal axis represents the number of buffers inserted.

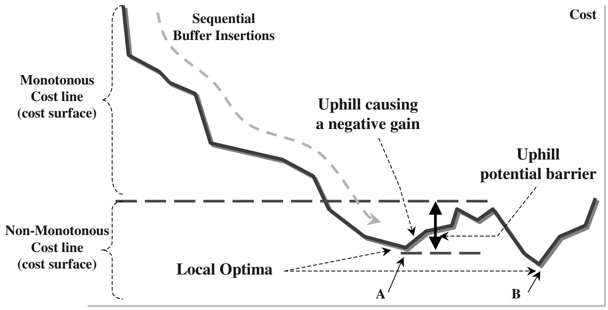
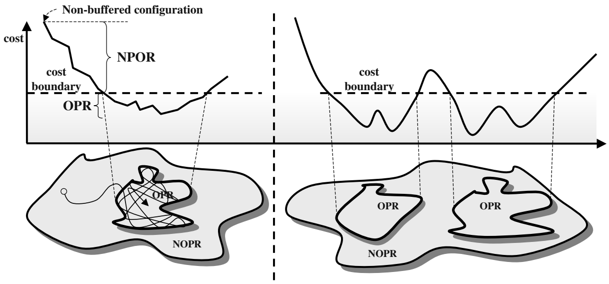


Fig. 7. Cost line and local optimums



(a) Bounding an OPR

(b) Too low boundary makes two OPRs

Fig. 8. Partitioning a design space

Definition 1. ***OPR*** *is a set of buffer configurations whose costs are lower than a given cost boundary. Between any two buffer configuration* BC_i *and* BC_j *of OPR, there exists at least one sequence of moves that passes through only buffer configurations in OPR*

Definition 2. ***NOPR*** *is a set of buffer configurations that are in a buffer design space but not in any OPR.*

As shown in Fig.8(a), the proposed search starts from a non-buffered configuration. In average-case optimized circuits, lots of buffers are allocated to each channel in order to reduce starvation/blocking effects caused by processing time variations. Thus, the non-buffered configuration is in NOPR in general. From the initial buffer configuration in NOPR, a search procedure based on single move decides the next buffer configuration simply by taking the biggest cost reduction till it reaches an optimum within an OPR. Once the search reaches OPR, a more refined search algorithm is executed within OPR.

The cost boundary between OPR and NOPR should be carefully selected since it plays a critical role in providing the optimality of search solutions as well as the efficiency of search. Selecting a very low cost boundary may result in a very small OPR. A small OPR means that only a small part of design space

```
Algorithm Buffer_Allocation
01: BC = 0;
02: while a negative gain does not appear in stat begin
03:     BC’ ← insert one buffer to the channel of BC that makes biggest cost reduction
04:     calculate the corresponding gain, cost(BC) – cost(BC’)
05:     BC = BC’
06:     store the gain to stat & maintain only latest n gains
07: end
08: LBC = BC’
09: derive a cost boundary using stat and cost(LBC)
10: OBC ← do simulated annealing with cost bound and LBC
11: return OBC;
```

Fig. 9. The proposed buffer allocation algorithm

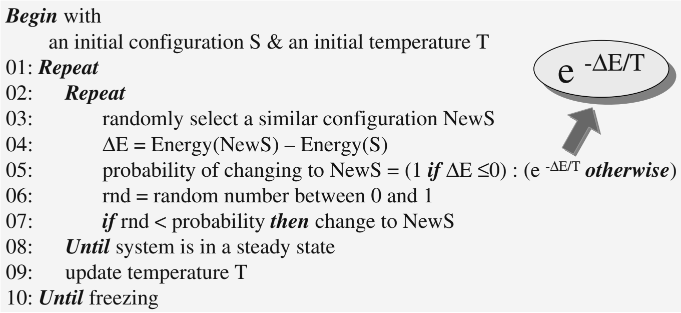
is explored with a high complexity search and hence high efficiency is expected. However, the boundary can yield multiple OPRs and causes a search to be trapped in an OPR that does not include a global optimum or a near optimum. Fig.8(b) shows a case of multiple OPRs due to a cost boundary that is too low. Since there is a possibility that an optimum or a near optimum does not belong to OPR the search procedure is exploring within, the quality of solution may deteriorate.

There are two ways to solve this problem. One is making a cost boundary high enough so that it does not yield multiple OPRs. However, a higher boundary increases the size of OPR and the search becomes inefficient. The other is allowing a probabilistic transition from one OPR to another even though such a transition increases cost over the given boundary. In this case, some moves enabling an escape from OPR are allowed with a small probability while most of the time the search process works mainly within the OPR. The escaping probability should be determined carefully as well.

5 An Efficient Buffer Allocation Algorithm

Fig.9 shows a top-level flow of the proposed buffer allocation algorithm. In Line 1, the initial buffer configuration BC is set. From Line 2 to Line 7, the search inserts a buffer sequentially to a channel selected for maximum cost reduction in each loop. The while-loop is repeated until negative gain appears. In Line 8, after completing the loop, *LBC*, for instance, corresponds to buffer configuration A in Fig.7. As shown in Fig.7, the cost boundary can be derived by adding the uphill potential barrier to the cost of *LBC* in our algorithm. The uphill potential barrier should be decided as small as possible while making it possible to reach other optima below the boundary cost.

Within the while loop, the gain obtained by each allocation is stored to stat temporarily in Line 6. Gain information in stat is used at Line 2 to check whether *BC* reached an OPR or not. The gain history is also used to derive a cost boundary in Line 9. After completing the loop, a target OPR is decided as a set of buffer configurations that are reachable from *LBC* without making a move that goes over the derived cost boundary.



```
Begin with
    an initial configuration S & an initial temperature T
01: Repeat
02:     Repeat
03:         randomly select a similar configuration NewS
04:         $\Delta$E = Energy(NewS) – Energy(S)
05:         probability of changing to NewS = (1 if $\Delta$E ≤0) : (e $^{-\Delta E/T}$ otherwise)
06:         rnd = random number between 0 and 1
07:         if rnd < probability then change to NewS
08:     Until system is in a steady state
09:     update temperature T
10: Until freezing
```

Fig. 10. An SA algorithm

Finally, in Line 10, a simulated annealing (SA) algorithm is executed so that the search is performed mainly within the target OPR with only a small escaping probability. LBC becomes an initial solution and the cost boundary is used for calculating an initial temperature for the SA. More detailed descriptions on the derivation of a cost boundary and initial temperature calculation for SA are as follows.

Cost Boundary Derivation : To make the algorithm efficient, a cost boundary should be chosen as small as possible. In this paper, we use a heuristic and experimentally validated cost boundary; cost(LBC) + D_{gain} where D_{gain} is the difference between the maximum and the minimum gain among the latest n gains in stat. The number n is set to the number of channels in an ALP.

D_{gain} can be seen as an approximation of the uphill potential barrier. A move is allowed only when a newly generated configuration by the move has a cost lower than cost(LBC) + D_{gain} once a high complexity search starts from LBC. Empirical results show that D_{gain} is a reasonable approximation of the maximum uphill potential barrier from LBC.

Applying Simulated Annealing : A traditional SA algorithm is presented in Fig.10. In simulated annealing, an energy function plays the same role as a cost function, hence in what we proposed, the energy function is implemented by the cost function defined in Section 3.2.

As mentioned earlier, LBC is used as an initial configuration for the SA. An initial temperature is derived using the cost boundary and the probability of escaping from an OPR. As shown in Line 5 - 7 of Fig.10, a decision on acceptance/rejection of a new configuration is made based on an energy function and a random number. If the new configuration decreases energy, then ΔE becomes negative and accepts the new configuration unconditionally. Otherwise, acceptance of an uphill move depends on the probability, $e^{-\Delta \mathrm{E}/\mathrm{T}}$. From the expression, $e^{-\Delta \mathrm{E}/\mathrm{T}}$, we can derive an initial temperature T using two parameters: the escaping probability from an OPR and an uphill potential barrier ΔE. ΔE is replaced by D_{gain} since the D_{gain} is considered as the maximum uphill potential that specifies the boundary of an OPR.

The escaping probability should be set to a small value in order to keep the boundary more or less strict. If escaping probability is set to 5%, an initial temperature is given as follows;

$$e^{-D_{gain}/\mathrm{T}} = 0.05 \quad \Longrightarrow \quad \mathrm{T} = \text{-}\ D_{gain}\ /\ \log(0.05)$$

Generally, an initial temperature for simulated annealing is set to the higher value than a standard deviation, σ, of cost values in a design space or $\frac{-3*\sigma}{lnP}$ where P denotes the initial probability of accepting solutions. The conventional value of P is 0.9 and thus $\frac{-3}{lnP}$ is about 20 [10,12].

6 Experiments and Discussion

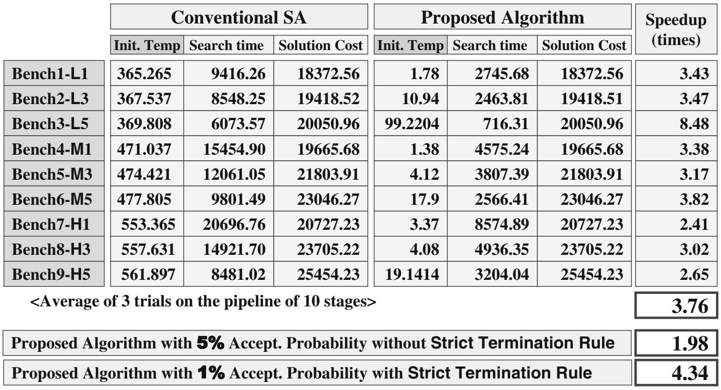
The proposed algorithm is implemented as a C++ program. Experiments are run on a 750MHz Sun Blade system. Since the previous work focused on mathematical analysis [5,9,11], there are no available benchmarks and comparable results. The test data in Table 1 are synthesized by varying “area of a buffer cell” and “range of variable processing delay in a stage” for an ALP consisting of ten stages. Each letter ‘L’, ‘M’, and ‘H’ in the name of test data corresponds to the delay range [10, 20], [5, 25], and [1, 29], respectively. The last single-digit in each test data name represents an area of a unit buffer cell. Three experiments are performed for each test data to eliminate a probabilistic variations of results by SA algorithm and the average over those three runs are presented in Table 1.

To show the effectiveness of the proposed method, we also show the results obtained by a conventional SA procedure in terms of solution quality and search time. As we mentioned in Section 4, ACTR is a proper metric for limiting the height of uphills to move from an OPR to another and was used to calculate an initial temperature for conventional SA. The initial temperature is calculated using a ‘maximum ACTR’ in a non-buffered configuration and ‘50% acceptance rate’. That is, ‘$Initial\_Temperature = \frac{Maximum\_ACTR}{log(0.5)}$’. Naturally, a non-buffered configuration is used as an initial configuration. When σ is evaluated on the test data, it is about two to four times higher than the ACTR-based initial temperature. When $\frac{-3*\sigma}{lnP}$ is used, temperature is over twenty times higher.

Except for the *initial temperature calculation* and the *initial configuration assignment*, the same cooling schedule is used for both conventional SA and the SA part of the proposed buffer allocation algorithm. Since there are many possible cooling schedules, overall execution times are variable. However, the initial temperatures are comparable between conventional and proposed SA. Consequently, the reduction rate of initial temperatures can be considered as a real contribution of this paper.

Over the synthesized test data, the proposed algorithm reduce the search time by 47% (1.98 speedup) in average while the optimality of solutions is kept when compared to the pure SA algorithm. Note that the reduction rate of the search time is not proportional to that of the temperature. This happens because that the number of iterations in the inner loop (Line 2 to Line 8 in Fig.10) of an SA algorithm tends to increase as temperature decreases.

Table 1. Experimental results



	Conventional SA			Proposed Algorithm			Speedup (times)
	Init. Temp	Search time	Solution Cost	Init. Temp	Search time	Solution Cost	
Bench1-L1	365.265	9416.26	18372.56	1.78	2745.68	18372.56	3.43
Bench2-L3	367.537	8548.25	19418.52	10.94	2463.81	19418.51	3.47
Bench3-L5	369.808	6073.57	20050.96	99.2204	716.31	20050.96	8.48
Bench4-M1	471.037	15454.90	19665.68	1.38	4575.24	19665.68	3.38
Bench5-M3	474.421	12061.05	21803.91	4.12	3807.39	21803.91	3.17
Bench6-M5	477.805	9801.49	23046.27	17.9	2566.41	23046.27	3.82
Bench7-H1	553.365	20696.76	20727.23	3.37	8574.89	20727.23	2.41
Bench8-H3	557.631	14921.70	23705.22	4.08	4936.35	23705.22	3.02
Bench9-H5	561.897	8481.02	25454.23	19.1414	3204.04	25454.23	2.65
<Average of 3 trials on the pipeline of 10 stages>							3.76
Proposed Algorithm with 5% Accept. Probability without Strict Termination Rule							1.98
Proposed Algorithm with 1% Accept. Probability with Strict Termination Rule							4.34

Thanks to “a low initial temperature at the beginning of the SA procedure” in the proposed algorithm and “a localized search space”, the SA procedure shows the fast convergence to a solution, compared with the conventional SA. The fast convergence allows further optimization of SA procedure. When the temperature is high, search behavior is unpredictable and the behavior shows divergence during the initial search. When the temperature is low, however, the repetitive oscillating configuration transitions (moves) can be thought as in the way of search convergence. Consequently, the proposed algorithm can further reduce the search time by employing a more restrictive steady-state checking rule (at Line 8 in Fig.10) in the SA algorithm without loss of optimality.

The proposed algorithm is modified to have a strict equilibrium steady-state checking rule limit *the number of iterations that does not introduce more cost reduction*. In this case, the proposed algorithm runs **3.76 times faster without loss of optimality** as shown in Table 1 when compared with the conventional SA without the strict termination rule. When we apply the same steady-state checking rule to the conventional SA, it resulted in significantly worse solutions due to the diverging behavior of a high temperature search although the search time is reduced.

To see the effects of a boundary escaping probability, more experiments are performed with 1% probability of escaping a boundary instead of 5%. In this case, a speedup ratio of search time reaches up to **4.34 without loss of optimality** as shown in the last row of Table 1. Consequently, we guess that the uphill potential barrier between optima is very small. Further analysis is required to find the lowest escaping probability from OPR while keeping the quality of solutions.

7 Summary and Conclusions

In this paper, we have analyzed a buffer design space of an ALP and developed an effective way to reduce the search space. In the proposed approach, a buffer design space is partitioned into two subspaces, an OPR and an NOPR, according to the characteristics (monotonicity, convexity) of the cost surface. Then a hybrid search algorithm has been devised. It performs a hill climbing search in an NOPR and high-complexity search in an OPR. As a high-complexity search method, an SA algorithm is adopted. The proposed algorithm shows a speed up of up to 4.34 times without the loss of optimality, when tested with synthetic test data, compared to a conventional SA algorithm.

Future work is under way to develop a new heuristic search procedure to be applied to OPR instead of SA since SA still takes a long time to allocate buffers into a pipeline of only ten stages. Finally, it will be also interesting to do mathematical or theoretical analyses on a buffer design space of an ALP showing a dynamic average-case performance behavior.

References

1. N.J. Nilsson, "Principles of Artificial Intelligence," *Springer-Verlag*, 1980.
2. T.E. Williams, "Analyzing and improving the latency and throughput performance of self-timed pipelines and rings," *In Proc. of International Symposium on Circuits and Systems*, pp.665-668, vol. 2, May 1992.
3. M.R. Greenstreet, "STARI: A Technique for High-Bandwidth Communication," *PhD. Thesis, Princeton University*, Jan. 1993.
4. S. Hauck, "Asynchronous Design Methodologies: an Overview," *In Proceedings of the IEEE*, vol.83, no.1, pp. 69-93, 1995.
5. D. Kearney, "Performance Evaluation of Asynchronous Logic Pipelines with Data Dependant Processing Delays," *In Proc. of the Second Working Conference on Asynchronous Design Methodologies*, pp.4-13, London, May. 1995.
6. D. Kearney and N.W. Bergmann, "Bundled Data Asynchronous Multipliers with Data Dependent Computation Times," *In Proc. of International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pp.186-197, Apr. 1997.
7. S.M. Nowick, K.Y. Yun et al., "Speculative completion for the design of high-performance asynchronous dynamic adders," *In Proc. of International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pp.210-223, Apr. 1997.
8. W.C. Chou, P.A. Beerel, et al., "Average-case optimized technology mapping of one-hot domino circuits," *In Proc. of International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pp.80-91, Apr. 1998.
9. D. Kearney, "Theoretical Limits on the Data Dependent Performance of Asynchronous Circuits," *In Proc. of International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pp.201-207, Apr. 1999.
10. S.M. Sait, H. Youssef, "Iterative Computer Algorithms with Applications in Engineering," *IEEE Computer Society Press*, 1999.
11. M.R. Greenstreet and B. Alwis, "How to Achieve Worst-Case Performance," *In Proc. of International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pp.206-216, Mar. 2001.
12. C.-C. Chang, J. Cong, Z. Pan and X. Yuan "Multilevel Global Placement With Congestion Control," *IEEE Tranc. on Computer-Aided Design of Integrated Circuits and Systems*, pp.395-409, Vol. 22, No. 4, Apr., 2003.

Author Index