Iowa State University
Digital Repository

Retrospective Theses and Dissertations | Iowa State University Capstones, Theses and Dissertations

2003

On-chip adaptive components for balanced computing

Rama Subba Reddy Sangireddy
Iowa State University

Follow this and additional works at: https://lib.dr.iastate.edu/rtd

Part of the Computer Sciences Commons, and the Electrical and Electronics Commons

Recommended Citation
Sangireddy, Rama Subba Reddy, "On-chip adaptive components for balanced computing " (2003). *Retrospective Theses and Dissertations*. 1459.
https://lib.dr.iastate.edu/rtd/1459

This Dissertation is brought to you for free and open access by the Iowa State University Capstones, Theses and Dissertations at Iowa State University Digital Repository. It has been accepted for inclusion in Retrospective Theses and Dissertations by an authorized administrator of Iowa State University Digital Repository. For more information, please contact digirep@iastate.edu.

المنارة للاستشارات
www.manaraa.com

On-chip adaptive components for balanced computing

by

Rama Subba Reddy Sangireddy

A dissertation submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Major: Computer Engineering

Program of Study Committee:
Arun K. Somani, Major Professor
Morris Chang
David Fernandez-Baca
Suraj Kothari
Akhilesh Tyagi

Iowa State University

Ames, Iowa

2003

Copyright © Rama Subba Reddy Sangireddy, 2003. All rights reserved.

UMI Number: 3105102

UMI®

UMI Microform 3105102

Copyright 2003 by ProQuest Information and Learning Company.

All rights reserved. This microform edition is protected against

unauthorized copying under Title 17, United States Code.

ProQuest Information and Learning Company
300 North Zeeb Road
P.O. Box 1346
Ann Arbor, MI 48106-1346

Graduate College
Iowa State University

This is to certify that the doctoral dissertation of

Rama Subba Reddy Sangireddy

has met the dissertation requirements of Iowa State University

Signature was redacted for privacy.

Major Professor

Signature was redacted for privacy.

For the Major Program

DEDICATION

To my mother
who works incessantly and untiringly to make our lives better, and
who stood strong and tall like a banyan tree as we flourished around her like vines.

TABLE OF CONTENTS

LIST OF FIGURES . . . vii

LIST OF TABLES . . . xii

ABSTRACT . . . xiv

1 INTRODUCTION . . . 1

1.1 Motivation . . . 4

1.2 Impact of Research . . . 6

1.3 Summary of Dissertation . . . 7

2 EFFECTIVE REGISTER FILE ORGANIZATION . . . 9

2.1 Introduction . . . 9

2.2 Register Lifetime Analysis . . . 11

2.2.1 Analysis Methodology . . . 13

2.3 TriBank Register File Organization . . . 17

2.3.1 Register Value Fetching to RF1 . . . 19

2.3.2 Freeing of Registers in RF2 and RF3 . . . 22

2.3.3 Impact on Bypass Logic . . . 24

2.3.4 Handling Precise Exceptions . . . 24

2.4 Performance Evaluation . . . 25

2.4.1 Simulation Methodology . . . 25

2.4.2 Results and Analysis . . . 27

2.5 Related Research . . . 31

2.6 Summary . . . 36

3 ADAPTIVE REGISTER FILE 38

3.1 Introduction 38

3.2 Related Research 39

3.3 Adaptive Register File Architecture 40

3.3.1 ARC as a Computing Unit 42

3.3.2 ARC as an Additional Register Bank 50

3.4 Performance Analysis 52

3.5 Summary and Future Work 55

4 LOW-POWER HIGH-PERFORMANCE ARCHITECTURES . . . 58

4.1 ABC Microprocessor 59

4.1.1 RFC Microarchitecture 60

4.1.2 Computing Structure in RFC Modules 63

4.1.3 Access Time and Energy Dissipation 67

4.2 Architecture Simulator for ABC Implementation 70

4.2.1 Instructions to Utilize RFC 70

4.3 Performance of ABC Processor 76

4.4 Power Estimation in ABC Processor 78

4.4.1 Power Estimation Models 80

4.4.2 Power Estimation in RFC 82

4.4.3 Results and Analysis 84

4.5 Summary 90

5 TIMING CONFIGURATION SWITCH 92

5.1 RFC Integrated Microarchitecture 93

5.1.1 Instructions to Utilize RFC 94

5.1.2 Mechanism for the Computation in RFC 95

5.2 Performance of ABC Architecture 98

5.3 RFC Configuration Schemes 101

5.3.1 Scheme1: One-time configuration of RFC 103

5.3.2 Scheme2: Continual configuration of RFC 103

5.4 Analysis of Execution Time 104
5.4.1 Execution Time Analysis With One-Time RFC Configuration . . 107
5.4.2 Execution Time Analysis With Continual RFC Configuration . . 109
5.4.3 Effect of Number of Core Function Instances (N) 110
5.4.4 Effect of Percentage of Core Function (P) 111
5.4.5 Effect of Cache Blocking Factor (ϕ) 111
5.5 Results and Analysis 111
5.6 Future Work 115
5.7 Summary 117
6 IP ADDRESS LOOKUP ENGINE 118
6.1 Introduction 118
6.2 Longest Prefix Matching 120
6.3 Related Research 122
6.3.1 *Binary Decision Diagrams* 125
6.4 BDD Based IP Address Lookups 126
6.4.1 Motivation 126
6.4.2 Details of the Scheme 126
6.4.3 *Reducing Effective Nodes* 127
6.4.4 Implementation Issues 130
6.5 Results and Analysis 131
6.5.1 Routing Table Update 132
6.5.2 Scalability to IPv6 136
6.6 Summary 137
7 CONCLUSIONS 139
7.1 Future Research 140
BIBLIOGRAPHY 142
ACKNOWLEDGMENTS 151
BIOGRAPHICAL SKETCH 154

LIST OF FIGURES

1.1 Bridging the gap. 3

2.1 Various stages in the lifetime of a physical register, for a particular mapping to a logical register. 12

2.2 Physical register lifetime distribution in (above) absolute number of cycles (below) percentage of lifetime, for SPEC 2000 integer benchmarks. 15

2.3 Physical register lifetime distribution in (above) absolute number of cycles (below) percentage of lifetime, for SPEC 2000 floating point benchmarks. 16

2.4 A TriBank Register file in the pipeline of the processor. 17

2.5 A TriBank Register file organization. 18

2.6 Example code 1. 21

2.7 Example code 2. 23

2.8 Instructions per cycle (IPC) throughput for various register file configurations in the 4-wide issue processor. 28

2.9 Instructions per cycle (IPC) throughput for various register file configurations in the 8-wide issue processor. 29

2.10 Relative instruction throughput when register access time is factored in for various register file configurations in the 4-wide issue processor. 30

2.11 Relative instruction throughput when register access time is factored in for various register file configurations in the 8-wide issue processor. 31

2.12 Percentage of run time during which at least one free register exists for various register file configurations in the 4-wide issue processor. 32

2.13 Percentage of run time during which at least one free register exists for various register file configurations in the 8-wide issue processor. 33

3.1 ARC unit placement in the processor pipeline. 40

3.2 32-bit 8-cycle Reconfigurable Matrix Multiplier. The write enable (WE) and clock signals are connected (not shown in figure) to all the LUTs. The multi-stage addition of partial products using CSAs is pipelined as shown. The end result is computed in a carry propagate adder (CPA). 44

3.3 Matrix multiplication in a ARC unit (shown excluding load/store and branch instructions). 46

3.4 32-bit 4-cycle Reconfigurable Matrix Multiplier. The write enable (WE) and clock signals are connected (not shown in figure) to all the LUTs. The pipelined addition of partial products using CSAs is performed as shown in Figure 3.2. 47

3.5 32-bit 2-cycle Reconfigurable Matrix Multiplier (partially shown). The write enable (WE) and clock signals are connected (not shown in figure) to all the LUTs. The pipelined addition of partial products using CSAs is performed as shown in Figure 3.2. . 48

3.6 A register sub-bank in the ARC unit. 51

3.7 Matrix multiplication in 4-wide out-of-order superscalar processor without ARC unit, and with ARC unit 54

3.8 Matrix multiplication in 8-wide out-of-order superscalar processor without ARC unit, and with ARC unit 55

3.9 Matrix multiplication in 16-wide out-of-order superscalar processor without ARC unit, and with ARC unit 56

4.1 ABC: RISC superscalar processor coupled with a 4-way RFC. . . 60

4.2 Data array structure in one module of A-way set associative RFC. The data array in other modules is similarly arranged 62

4.3 One tap of FIR filter implemented in four rows of LUTs in an RFC module with 8 columns of LUTs. The blank LUTs do not participate in computation mode, but function normally in memory mode. 65

4.4 Two successive processing elements (PEs) for DCT/IDCT, implemented in four rows of LUTs in an RFC module with 8 columns of LUTs. The blank LUTs do not participate in computation mode, but function normally in memory mode. 66

4.5 Access time, at 0.8 micron CMOS technology, in conventional cache and the RFC for Convolution algorithm implementation. x-axis indicates the cache associativity from 2- to 32-way, for each cache size in KB. 68

4.6 Energy dissipation, at 0.8 micron CMOS technology, in conventional cache and the RFC for Convolution algorithm implementation. x-axis indicates the cache associativity from 2- to 32-way, for each cache size in KB. 69

4.7 rfc instructions for loading and storing "word" type of data . . . 71

4.8 State transition for the RFC cache. 73

4.9 Performance of ABC processor vs. base processor for a 32KB 4-way cache with a line size of 32B 78

4.10 Component power utilization in ABC processor vs. base processor for MPEG2 decode application. 85

4.11 Component power utilization in ABC processor vs. base processor for MPEG2 encode application. 86

4.12 Component power utilization in ABC processor vs. base processor for FIR application. 87

4.13 Component power utilization in ABC processor vs. base processor for cjpeg application. 88

4.14 Component power utilization in ABC processor vs. base processor for IIR application. 89

4.15 Total power utilization in ABC processor vs. base processor using (a) Alpha processor model (b) Pentium-Pro model. 90

5.1 Reconfigurable Functional Cache (RFC) organizations and address mapping with (a) 4 cache modules (b) 16 cache modules. . 93

5.2 Normalized execution cycles in base processor without RFC and ABC processor with RFC, with varying cache organizations. . . 99

5.3 Distribution of core functions in MPEG applications. 102

5.4 Variation of cache blocking factor with the fraction of core function. 108

5.5 Variation of cache blocking factor with speed-up in core function. 109

5.6 Normalized execution cycles in the base processor without RFC, and the ABC processor with different RFC configuration schemes. 113

5.7 Relative performance improvement in two RFC configuration schemes. 114

5.8 Architectural Design Space. 116

6.1 Packet routing based on longest prefix matching mechanism. . . 121

6.2 Function $f = x_0x_1 + x_1x_2 + x_2x_0$ represented as (a) Binary decision tree and (b) BDD. 125

6.3 Binary decision tree for the sample routing table. Dotted nodes are redundant. 127

6.4 Binary decision tree for (a) NHP_1 (b) NHP_0, with all effective nodes assigned with output. Dotted nodes are redundant. 128

6.5 BDDs for (a)NHP_1. (b)NHP_0 128

6.6 (a) Nodes with assigned NHP ports. (b),(c),(d),(e) Output bits assigned to each of the nodes in 4-bit binary encoding of NHP. Dotted nodes are redundant. 129

6.7 CLB mapping in FPGA. 131

6.8 (a) Binary decision tree representation of the modified routing table. Dotted nodes are redundant. (b) Modified BDD for NHP_0. 134

6.9 Aggregatable Global Unicast Address for IPv6. 137

LIST OF TABLES

2.1 Simplescalar simulation parameters. 14

2.2 Configurations for various register file organizations simulated. A bus from RF2 to RF3 indicates additional one read port for RF2 and one write port for RF3. Access time is measured at 0.18μ. 26

3.1 Simplescalar simulation parameters. 41

3.2 On-chip area for ARC design using 3-LUTs, implemented at 0.18μ technology. Area overhead is ARC unit area excluding LUTs (registers). 50

3.3 Register access time in the ARC unit at 0.18μ technology. 51

3.4 Various processor configurations simulated. 52

4.1 Comparative performance results of ABC processor and base processor for various benchmarks, for a 32KB 4-way cache with a line size of 32B 76

4.2 Listing of heuristic power estimations for Alpha processor model 81

4.3 Listing of heuristic power estimations for Pentium-Pro processor model 82

5.1 Architecture Design Parameters. 105

5.2 Variation in parameters for various RFC configuration schemes. . 107

6.1 A sample routing table. 120

6.2 Data throughput and packet processing time budgets for ATM over SONET. Packet size considered is 40 bytes. 122

6.3	Effective nodes for sample routing tables and the real-time 32-bit IP MAE-east routing table with 24792 prefixes.	130
6.4	Lookup time performance analysis of BDD based routing engine. Throughput is number of packets per second.	132
6.5	Modified routing table.	133
6.6	Number of corresponding nodes in each level of binary decision trees that differ in their output. The two binary decision trees compared are for adjacent snapshots of real-time MAE-east routing table.	135

ABSTRACT

The demand for higher computing power to effectively execute compute-intensive functions and thus more on-chip computing resources is ever increasing. On the other hand, design of an effective register file architecture is becoming a bottleneck for meeting the memory-bandwidth demand of modern wide-issue dynamically scheduled superscalar processors. A need for a balance in the memory-bandwidth and the computing rate is significant to achieve a higher processor throughput. Further, at the advent of mobile and ad-hoc computing, processors are being expected to consume lesser amounts of energy even while delivering higher performance. This dissertation aims to address the above issues in the context of reconfigurable architectures, with the underlying concept of utilizing on-chip components for memory or computing purposes depending on the demand from an application. To efficiently utilize silicon real-estate on the chip, we exploit the possibility of using on-chip memory elements as computing units.

First, this dissertation proposes *TriBank Register file* architecture, a novel register file organization for wide-issue dynamically scheduled superscalar processors. The organization exploits long latencies in the lifetime of a register to meet the two requirements of a small register access time and a large memory bandwidth. Implementation of the TriBank register file organization, as compared to a conventional monolithic register file in an 8-wide out-of-order issue superscalar processor enhanced the throughput in instructions per cycle (IPC) by 3% and 14%, for SpecInt2000 and SpecFP2000, respectively. When the register file access time is factored in, the instruction throughput is enhanced up to 56% and 96%, for SpecInt2000 and SpecFP2000, respectively. The significant contribution of our proposed register file architecture is that it enhances IPC, when earlier work in designing effective register file has resulted in IPC degradation.

Next, the dissertation proposes *Adaptive Register File Computing (ARC)* unit, a novel on-chip processing element that leverages application-specific processing capabilities. The ARC unit supplements a conventional register file to provide large memory bandwidth, or acts as a configurable computing unit to provide higher on-chip computing capacity, depending on the requirement of a specific application. When an out-of-order 8-wide issue superscalar processor is supplemented with the ARC unit to process matrix multiplication, a compute-intensive core function in most multimedia applications, results show a performance increase of up to 12%. Similarly, a 17% performance enhancement is seen when the matrix multiplication is performed in an out-of-order 16-wide issue superscalar processor supplemented with the ARC unit. The dissertation also discusses the microarchitecture level details for the implementation of the ARC unit.

The dissertation also explores the ability of the reconfigurable computing models in delivering high performance while providing with significant savings in the energy dissipation in the various on-chip components of the processor. For the Reconfigurable Computing Cache (RFC) based processor, developed earlier to utilize a module of an L1 data cache is used as a coprocessor to process compute intensive multimedia applications, the impact of RFC on cache access time and energy dissipation has been explored. We show that reduced number of cache accesses and lesser utilization of other on-chip resources, due to a significant reduction in execution time of application, will result in energy savings. The results show that up to 60% reduction in power consumption is achieved for MPEG decoding, and a reduction in the range of 10% to 20% for various other multimedia applications. This dissertation further explores the issues of management of RFC, where the impact of various schemes for configuration of core function into the RFC module is studied. It also gives a detailed analysis on the performance of the RFC based processor in terms of the execution time of application for various configuration schemes, including the study of the effect of the percentage of the core function in an entire application over the management of RFC modules.

1 INTRODUCTION

Multimedia and digital signal processing applications demand more and more computing power. The widely known 90-10 rule predicates that 90% of the execution time is expended by about 10% of the application code which is compute intensive and that the remaining 10% of the execution time is consumed by inner loops in general. The spatial structures excel in the execution of such compute intensive functions as compared to the temporal structures [19]. The spatial structures help overcome the key performance bottleneck of processor-memory communication bandwidth gap as experienced in the general purpose microprocessors.

Rather than juggling the intermediate results in and out of registers and memory, the spatial compute engine is customized so that the data flows directly from source to sink. For example, a spatial implementation of a simple filter, takes in a new sample and computes a new result in a single cycle. On the contrary, a temporal structure like a general purpose processor or a DSP takes a few cycles to evaluate each filter tap, easily running tens of cycles for even the simplest filter structures. Further, the reconfigurability of such spatial structures provides flexibility to the system for executing a wide range of compute intensive functions. The time for loading the configuration of a particular function can be amortized over long execution time and hence can be offset by the speed-up obtained. On the other hand, it is to be noted that the reconfigurable devices get bogged down on the large portions of the code that are rarely executed with repetition and hence loading the configuration for every small segment of computation forms the bottleneck. Hence a practical compromise is to couple a reconfigurable device with a conventional processor in order to exploit the strengths and overcome the limitations of each.

Subsequently, the concept of a general purpose processor with a tightly coupled reconfigurable logic arrays has been widely recognized as the main focus for the development of future computing systems. To meet the increasing demand for more computing power, the technology of implementing a compute intensive function in a single reconfigurable hardware unit has evolved to accelerate the execution of selective functions. Such reconfigurable chips are used as coprocessors in tandem with the general purpose processor. In such hybrid computing models, it is necessary that the reconfigurable logic be subservient to the traditional processor, which will be the master processor, for at least two reasons: (1) the traditional processor executes the control code which logically binds together the various computations the reconfigurable device will perform, and (2) execution on the traditional processor then becomes the default condition, thus making the hybrid machine compatible to the existing computing practice.

In the past decade, the research community in the area of *Reconfigurable Computing* has built various models based on the following principles: (1) Reconfigurable devices may do well with compute intensive functions in applications, but traditional processors are still superior in the execution of remaining code that is irregular and/or rarely repeated. (2) For reconfigurable computing to be economically viable, the various parts included in the design have to be squeezed down to one chip. (3) Reconfigurable hardware will not meet its full potential unless reconfiguration time is minimized and the reconfigurable hardware also has access to the system's memory that is at least as good as that available to the main processor. For an efficient utilization of the reconfigurable hardware in the system, the essential characteristic of all the computing models is that some amount of state is semi-static, i.e., it changes frequently enough to take advantage of programmability but slow enough to mask the hardware configuration time.

The reconfigurable computing architectures have been developed to exploit the performance of the customized hardware units and the flexibility of the traditional processors. Conventional general purpose processors and the custom computing hardware units live at the extreme ends of a rich architectural space, as shown in Figure 1.1. The reconfigurable platforms are heading from niche to mainstream, bridging the gap

between ASICs and the microprocessor.

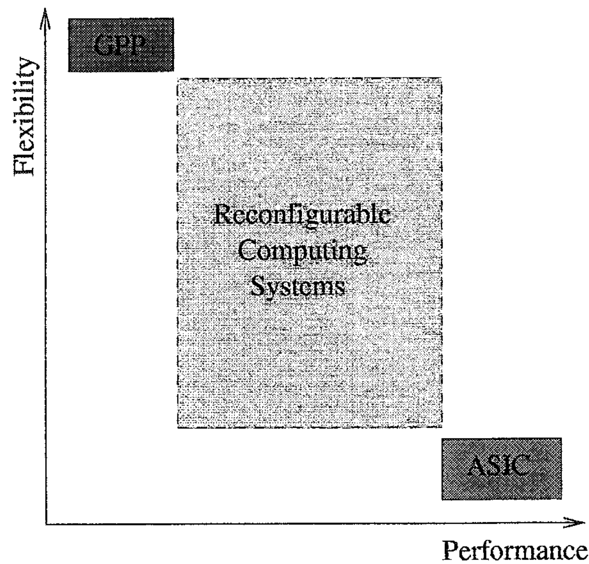


Figure 1.1 Bridging the gap.

Exploding design cost and shrinking product life cycles of ASICs create a demand on reconfigurable logic array usage for product longevity. Performance is only one part of the purpose of development of the reconfigurable computing models. The need is also there to fully exploit the flexibility of the adaptive hardware. Many system-level integrated future products without reconfigurability will not be competitive [29]. Instead of the technology progress, better architectures using reconfigurable arrays is the key to keep up the current innovation rate beyond the limits of the silicon. The commercial acceptability of a computing model mainly depends on its ability to execute various applications that would be developed even after the design of the computing model. In other words, the flexibility of the current computing models to adopt various future applications will determine their survivability. Hence, the reconfigurable hardware, integrated with the main processor to accelerate the compute intensive functions, has to be designed with a certain degree of flexibility to implement a wide range of special functions.

Contrary to the usage of FPGAs, the popular reconfigurable devices that provide fine grain reconfigurability, the field of reconfigurable computing prefers the use of coarse grain reconfigurable arrays with datapath width greater than one bit. This is due to the fact that the fine-grained architectures are much less efficient due to a huge routing area overhead and poor routability. Thus, to accelerate structured and regular computations, such as DSP and multimedia applications, a reconfigurable logic that is specialized for these computations has been developed and integrated into on-chip microprocessors [30, 85, 17, 59, 81]. This kind of customized computing units could produce high performance for those computations. However, the on-chip reconfigurable logic may result in relatively low utilization of the on-chip resources when only a few computations exploit the logic.

One of the factors that is significant to consider, while building an efficient computing model, is that it is important to ensure that all the hardware resources available on the chip be utilized to the maximum extent possible for wide range of applications. For example, current processor designs often devote a largest fraction (up to 80%) of on-chip transistors to caches. However, many workloads, like the media processor applications, do not fully utilize the large cache resources due to the streaming nature and the lack of temporal locality for the data. On the contrary, these multimedia applications require more computing resources due to the fact that they primarily are compute intensive functions.

1.1 Motivation

The increase in performance of the memory system is unable to meet the phenomenal increase in the microprocessor performance [31]. Hence computer designers continuously face the challenge of bridging this increasing processor-memory performance gap, to improve the computer system performance. An ideal memory system, that can serve the CPU requests instantaneously, is impractical to implement as the three factors of memory (capacity, speed, and cost) are in contraposition. To alleviate this problem, smaller faster memories are placed on-chip closer to the central processing unit (CPU), at a moderate cost, to intercept and exploit locality of requests to larger, slower, and

cheaper memories.

Modern general-purpose computers contain registers, one or two levels of cache memory, main memory, and virtual memory, in that order along the path of CPU's request for data. With the advent of dynamically scheduled wide-issue out-of-order superscalar processors, the demand for high-performance memory system is growing. Microprocessor designers exert to increase the issue width to achieve high performance. Wider issue processors require more ports in the register file which has an adverse affect on the register access time. Besides, a wide-issue superscalar processor is effective only if as many instructions as possible are issued each cycle, which implies that the processor has to view a larger instruction window to achieve sufficient amounts of instruction level parallelism (ILP). Large instruction window implies a larger set of in-flight instructions requiring a larger number of physical registers [25]. However, increasing the size of register file adversely affects the register access time.

A task when computed at certain processing rate (instructions per second) requires a certain memory bandwidth (bytes per second). The memory bandwidth requirements depend on the size and organization of the on-chip memory (registers and caches). A computation is said to be *balanced*, if the memory bandwidth of the processor exactly matches the memory bandwidth requirements of the computation with a fixed local memory configuration [44]. In this context, all the three parameters in the system: computation rate, on-chip memory configuration, and memory bandwidth, are fixed. It is not feasible to achieve balanced computation for most general purpose computations using such fixed processor configuration. However, if the parameters in consideration are dynamically variable it is possible to push the system towards a balanced computation and achieve a higher throughput (instructions per second).

Based on the above observations, the dissertation proposes *Adaptive Register file Architecture*, a novel architecture for dynamic superscalar processors that strives towards achieving the three requirements for greater performance: larger memory bandwidth, higher on-chip computing power, and balanced computing. The development of an *Adaptive Register file Architecture* first constitutes the design of a register file organiza-

tion that effectively meets the demand for large memory bandwidth in dynamic superscalar processors. The next phase involves the design of a register file that can act as a computing unit when higher on-chip computing power is demanded by an application.

Keeping in pace with the current requirement of power-aware architectures, the dissertation also explores the ability of the reconfigurable computing models in delivering high performance while providing with significant savings in the energy dissipation in the various on-chip components of the processor. Further, in continuance of our effort to build an efficient reconfigurable architecture with minimum configuration overhead, various configuration schemes have been studied. A detailed study of the performance analysis of the architecture, in terms of the execution time of the application, is given.

The impact of various architectural parameters and the factors governing the structure of an application over the execution time of an application has been extensively studied. With the help of the study undertaken, the design of reconfigurable architecture can be incorporated with the dynamic decision capability, so that appropriate configuration scheme is chosen dynamically for running a particular application. The dissertation also delves into the possibility of targeting compute intensive applications in the area of computer communications, to be computed in reconfigurable elements. The research discusses in detail the problem of address lookups in IP domain for the fast routing of communication packets, that forms a bottleneck in designing efficient network processors. A reconfigurable hardware based solution is proposed to deliver high performance.

1.2 Impact of Research

As the designers strive continuously to enhance the performance of dynamic superscalar processors by adding new structures or modifying the existing components on the chip, the complexity of the processor has been increasing. A wide spread research is currently underway to effectively address this increasing complexity in dynamic superscalar processors to scale with the technology and other influencing factors. One of the primary goals of this dissertation is to design an effective register file organization

for dynamic superscalar processors. The dissertation first performs a quantitative and qualitative analysis of the lifetime of a physical register in a superscalar processor, before proceeding to design an effective register file organization.

The dissertation further explores the idea of using various on-chip memory components for computational purposes. The dissertation aims to show that applications depending on their nature, demand either higher computing capacity or larger memory bandwidth or both. Hence, providing memory and computing resources on-chip that are fixed in nature is expensive and does not enable an efficient utilization of silicon real estate on-chip. Instead, it is more beneficial to design a portion of on-chip resources to be reconfigurable, so that it can be used either as a memory element or a computing unit, as the situation demands. For the purpose, the dissertation developed *Adaptive Register file* architecture, a novel architecture to address the above problems concurrently. The *Adaptive Register file Computing* (ARC) unit provides a higher on-chip computing capacity by executing a compute-intensive function, and provides a larger register file resources to meet the memory bandwidth requirements.

1.3 Summary of Dissertation

This dissertation is organized as follows.

Chapter 2 This chapter presents the design of an effective register file organization for dynamic superscalar processors.

Chapter 3 In this chapter we discuss in detail the design of an Adaptive register file computing unit, that provides higher on-chip computing capacity when required.

Chapter 4 In this chapter we discuss the ability of reconfigurable architectures in delivering higher performance while resulting in lesser power consumption.

Chapter 5 This chapter presents a model and analysis of timing a configuration switch to enable the design of an efficient reconfigurable computing architecture.

Chapter 6 The IP address lookup problem, a compute intensive application that forms a bottleneck in the performance of network processors is discussed in detail in this chapter. A reconfigurable computation based hardware design for the problem is proposed and thus the application is targeted for computing in a reconfigurable coprocessor.

Chapter 7 The conclusions for the dissertation, along with the directions for future research, are presented in this chapter.

The review of the earlier work has been discussed in each of the chapters as relevant to the issues addressed.

2 EFFECTIVE REGISTER FILE ORGANIZATION

2.1 Introduction

The increase in performance of the memory system is unable to compete with the phenomenal increase in the microprocessor performance [31]. Hence computer designers continuously face the challenge of bridging this increasing processor-memory performance gap to improve the overall computer system performance. An ideal memory system, that can serve the CPU requests instantaneously, is impractical to implement as the three factors of memory (capacity, speed, and cost) are in contra-position. To alleviate this problem, smaller faster memories are placed on-chip closer to the central processing unit (CPU), at a moderate cost, to intercept and exploit locality of requests to larger, slower, and cheaper memories. Modern general-purpose computers contain registers, one or two levels of cache memory, main memory, and virtual memory, in that order to service the CPU's request for data. With the advent of dynamically scheduled wide-issue out-of-order superscalar processors, the demand for high-performance memory system is further growing. Wider issue processors require more ports in the register file which has an adverse affect on the register access time. Besides, a wide-issue superscalar processor is effective only if as many instructions as possible are issued during each cycle, which implies that the processor has to view a larger instruction window to achieve sufficient amounts of instruction level parallelism (ILP). Large instruction window implies a larger set of in-flight instructions requiring a larger number of physical registers. However, increasing the size of register file adversely affects the register access time.

Register access time, along with time delays in issue window, dispatch stage, and

bypass logic in the processor pipeline mainly contributes toward designing the processor cycle time [56]. As the issue width of the processor increases, the complexity of each of these components increases, and a wide scale effort is underway to reduce the complexity and its effects in the processor design. Farkas *et al* [25] have shown that an 8-wide issue superscalar processor handling precise exceptions increases the average instruction throughput (IPC) as the register file size is increased up to 256. Besides, the portion of execution time during which there is at least one free physical register available, for logical register renaming at dispatch stage, increases considerably as the register file size increases. This is important for dispatching as many instructions as possible to instruction window to draw more ILP. However, the processor loses performance in terms of average number of billion instructions per second (BIPS) for a register file size beyond 128. This happens due to the adverse impact of the large register file access time on the processor cycle time. On the other hand, Lebeck *et al* have proposed a fast instruction issue window assisted by a large wait instruction buffer (WIB) of size up to 2K entries for extracting higher amounts of instruction level parallelism (ILP) [47]. For such a large instruction window size to be utilized to the maximum extent possible, it is necessary that as many instructions as possible are renamed at the dispatched stage. This is possible only with an availability of a large number of physical registers. Hence, design of an efficient register file, with as many registers as possible, but with a smaller register access time, is essential for an effective superscalar processor design.

Using the above observations, we develop a *TriBank Register file*, a register file architecture that performs well in meeting the following two main goals:

* Provide a small register access time to enable a faster processor cycle time and thus enhance the throughput of number of instructions per second (IPS).

* Provide a large number of registers to enable dispatching as many instructions as possible to issue window for extracting higher ILP, thus enhancing the throughput of number of instructions per cycle (IPC).

The above two goals are met by designing a register file that exploits long latencies

involved, in between allocation of register to a logical value and actual consumption of the value by a functional unit, and then in between consumption of the value and actual freeing of physical register for next allocation.

The rest of the chapter is organized as follows. In Section II, we provide a detailed analysis of the register life time as a precursor for our proposed design. In Section III, we present a detailed design of the proposed register file organization. The section also discusses the necessary modifications in the microarchitecture. Section IV analyzes the performance of the architecture. Section V discusses the related work done. Section VI concludes the chapter.

2.2 Register Lifetime Analysis

While undertaking the task of designing an effective register file for next generation superscalar processors, first we study and analyze the activity of a physical register during its lifetime one logical to physical mapping. For this purpose, we consider a DEC Alpha 21264 processor [39] based microarchitecture. The Alpha 21264 processor consists of a deep pipeline with fetching and renaming of instructions performed in-order. When the source operands of an instruction waiting in issue stage are ready, the instruction is issued for execution. Once granted execution, the register tags of source operands of the instruction are used to access the register file in the register read stage of the pipeline. The operand values read from the register file are forwarded to the appropriate functional unit in the execute stage of the pipeline. If a dependent operation is issued for execution immediately following the current instruction, the dependent instruction will read a stale value from the physical register file. A bypass logic is provided in the execute stage to select between the incoming register operand, or a more recent value on the bypass bus. Dependent instructions that execute in subsequent cycles must communicate via the bypass bus. All other instructions communicate through the physical register file.

The logical destination register for an instruction fetched is mapped to a free physical register at the dispatch stage. Subsequent instructions with the same logical register as their source operand are assigned to read from the mapped physical register. The logical

to physical register mapping remains active until another instruction with same logical register as its destination enters the dispatch stage. The logical destination of that instruction is then mapped to another free physical register and the process continues. The dispatch stage in the pipeline is stalled when no free physical register is available. This scheme of logical to physical register mapping eliminates the write after read (WAR) and write after write (WAW) data dependencies. The earlier allocated physical register is freed only when the subsequent instruction with same logical destination is committed. This is done to enable the recovery from branch mis-predictions and handle exceptions precisely. The conditions for freeing registers are described in more detail in [25].

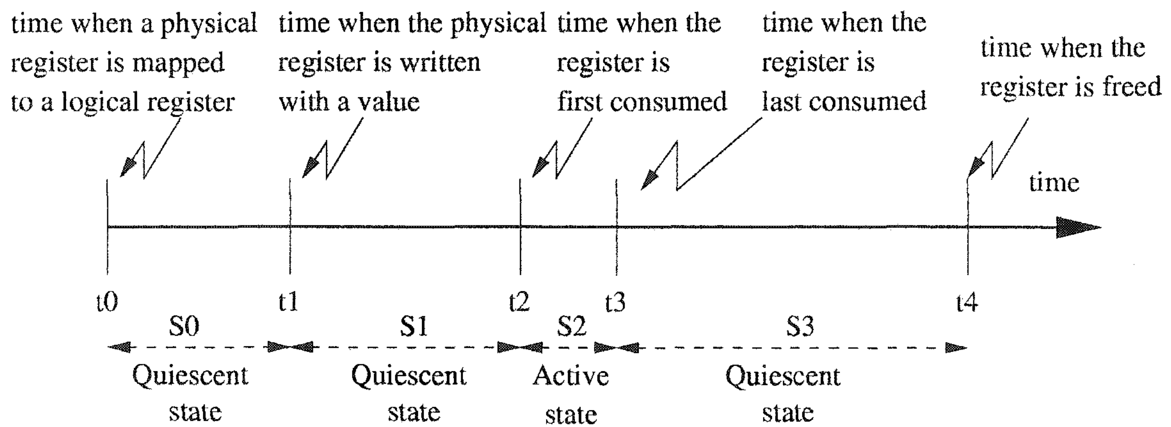


Figure 2.1 Various stages in the lifetime of a physical register, for a particular mapping to a logical register.

The life cycle of a physical register is identified as the time between its allocation to a logical destination at the dispatch stage and the time when it is freed. The various stages in the register lifetime, as illustrated in Figure 2.1, are:

t0: Time at which a free physical register R_p is allocated to a logical destination register R_l of an instruction I_k at the dispatch stage in the processor pipeline.

t1: Time at which R_p is written with a value. This happens when the instruction I_k is in the writeback stage.

t2: Time at which the value in R_p is first consumed. This happens when an instruction I_{k+l}, with a logical source operand of R_l, is issued for execution.

t3: Time at which the value in R_p is consumed for the last time. This happens when an instruction I_{k+m} $(m > l)$, with a logical source operand of R_l, is issued for execution and no further instructions use R_l as source operand until the R_l becomes a destination register for another instruction I_{k+n}.

t4: Time at which the physical register R_p is freed and is ready for the next allocation. This happens when instruction I_{k+n}, with logical destination R_l, is committed.

A conventional monolithic register file maintains the mapping of the physical register throughout the life cycle of each logical to physical mapping. Note that at the microarchitecture level it is easy to identify when a register value is first consumed. However, it is not easy to determine when it is consumed for the last time. To do so, it requires a large overhead of keeping track of all the instructions that are potential customers for the operand using some sort of counting mechanism to track the instructions as and when they are executed. In this study, we first identify the time of first and last consumption of a register value for only the purpose of analyzing the register activity during lifetime of its allocation to a logical register. Using that next we develop our architecture where we do not have to keep track of time of the first and last consumption of a register.

2.2.1 Analysis Methodology

To study a relationship among these various times, we used Simplescalar-3.0 [9] for the Alpha AXP instruction set to simulate a dynamically scheduled out-of-order issue superscalar processor with the simulation parameters depicted in Table 3.1. The instructions are traced along the various stages of the processor pipeline and the time intervals between various stages in the lifetime of a register are measured according to the above mentioned specifications. The time intervals measured are:

t1-t0: Time during which the register is waiting for the result to be written into it after it is allocated.

t2-t1: Time during which the register is waiting to be read by a functional unit after it is written into.

t3-t2: Time during which the register is active as supplier of an operand to functional units.

t4-t3: Time during which the register is waiting to be freed after it is consumed for the last time.

Table 2.1 Simplescalar simulation parameters.

Parameter	*Value*
Instruction cache *- latency*	32KB, 2-way, 32B line 1 cycle
Data cache *- latency*	32KB, 4-way, 32B line 1 cycle
Branch predictor $-mis-prediction\ latency$ $-return\ address\ stack\ size$	bimodal, 2K table size 7 cycles 8
Instruction issue queue size	64 (INT and FP, each)
Load/store queue (LSQ) size	128
ReOrder buffer (ROB) size	128
Issue width	2/4/8/16
Commit width	2/4/8/16
Functional units *–Integer arithmetic* *–Integer multiplier* *–floating point arithmetic* *–floating point multiplier*	 8 4 4 4
L2 unified cache *–latency*	256KB, 4-way, 64B line 6 cycles
TLB $-D-TLB$ $-I-TLB$ *–latency*	 512KB, 128 entries 256KB, 64 entries 30 cycles
Memory *–latency first, next* *–bus width*	 70, 2 cycles 8B

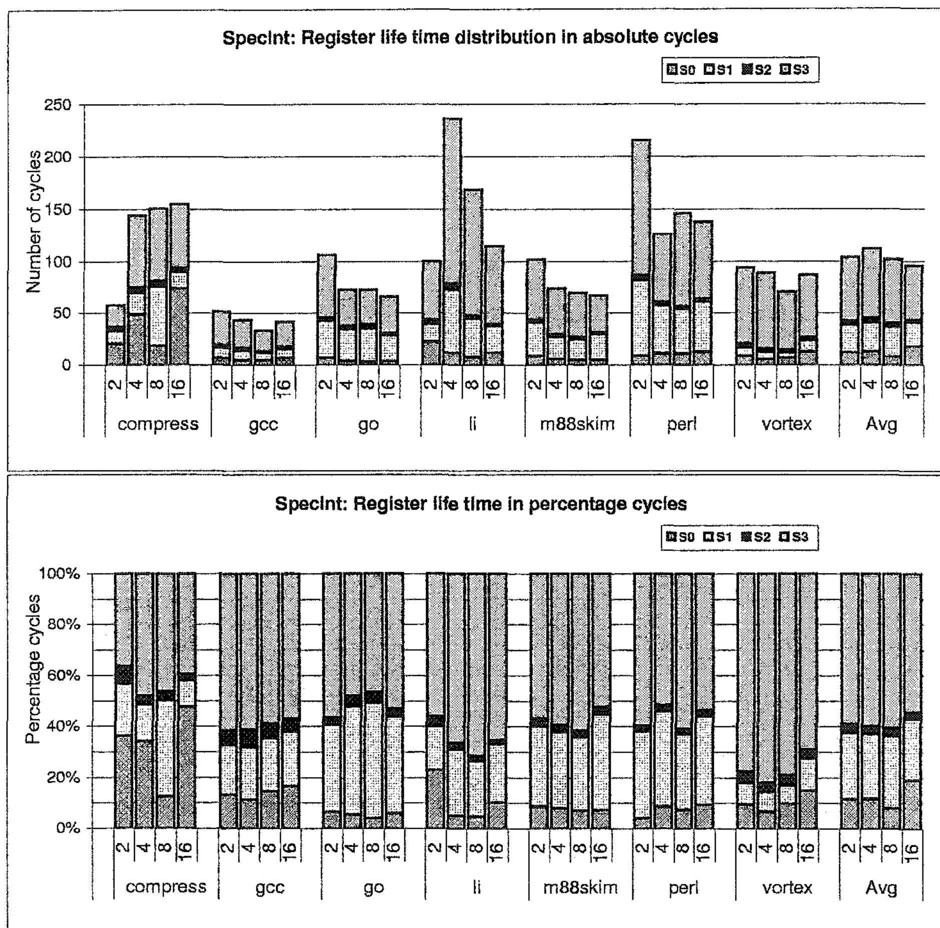


Figure 2.2 Physical register lifetime distribution in (above) absolute number of cycles (below) percentage of lifetime, for SPEC 2000 integer benchmarks.

The average time interval between each stage of register lifetime is shown in Figures 2.2 and 2.3 for various Spec integer and floating point benchmarks, respectively. The upper chart in each figure shows the register lifetime in absolute number of cycles and the lower chart illustrates the same in terms of percentage of time for each interval. The benchmark programs are simulated on the superscalar processor with various issue widths. The register lifetime can be classified, as illustrated in Figure 2.1, into an active state (S2) where the register is supplying the values to functional units, and quiescent states (S0, S1, and S3) where the physical register is inactive waiting for some action to

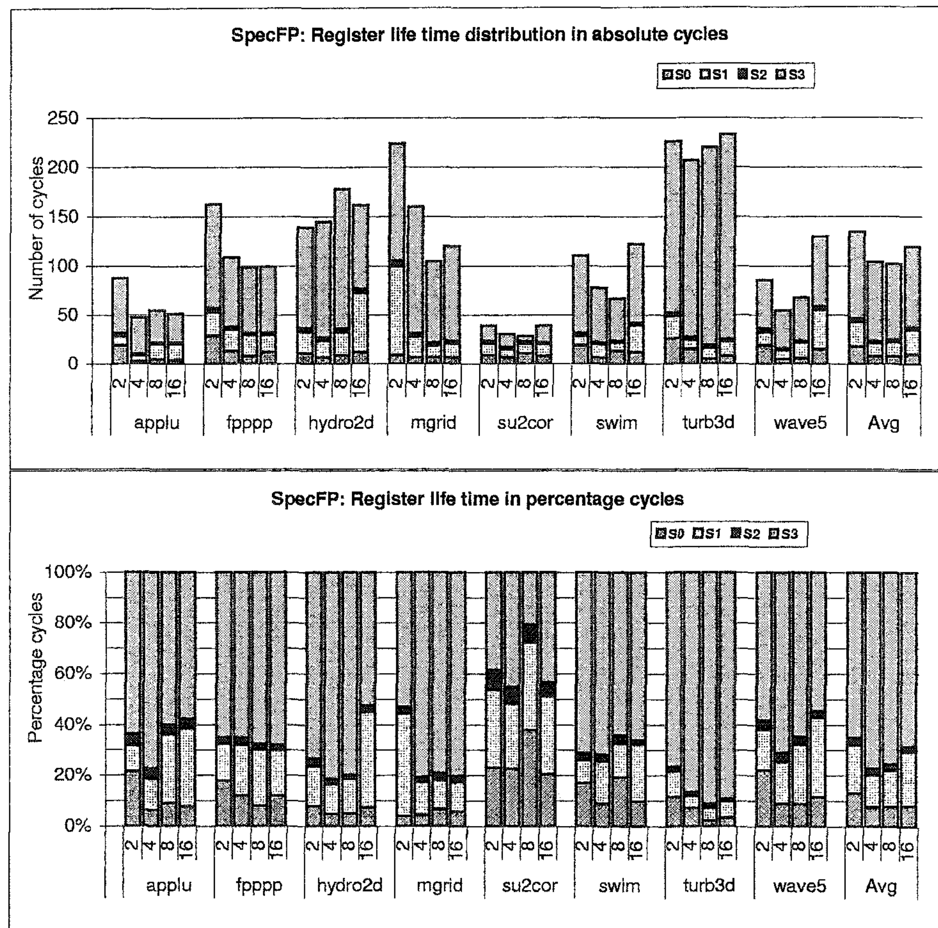


Figure 2.3 Physical register lifetime distribution in (above) absolute number of cycles (below) percentage of lifetime, for SPEC 2000 floating point benchmarks.

take place. It can be observed that the average active time of the register is exceptionally small (around 1% to 5%). This is mainly due to two reasons:

1. It is observed that around 85% of the time a register value is read at most once.
2. Some registers are never read as the value they hold are either supplied to their consumers through the bypass logic or even not read at all.

The observations that emanate from the above analysis are:

* Physical registers are allocated at dispatch stage, early in the pipeline, and expe-

rience a long latency before consumption.

* Amount of duration when the register is an active supplier of values to consumers is very small as compared to its long lifetime.

* In a large register file the number of registers that are active suppliers of data operands at a given time is small.

* After the last consumption by a functional unit, there is a long latency before a register is freed up.

* A more aggressive logical to physical mapping at dispatch stage can be obtained by hiding the latency in freeing of registers, with the support of a mechanism to handle precise exceptions.

2.3 TriBank Register File Organization

The above observations lead us to design a TriBank register file architecture. Our proposed architecture is shown in Figure 2.4. The TriBank Register file organization consists of three banks of physical registers with a heterogeneous structure, as shown in Figure 2.5. Each register bank consists of a different number of registers and different number of ports according to the architecture requirements as discussed later. The function of each register bank is described below.

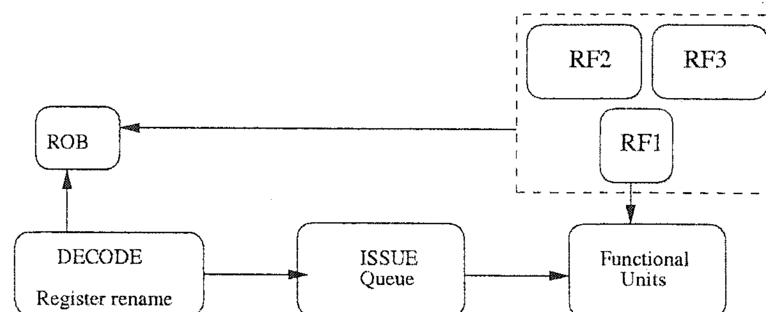


Figure 2.4 A TriBank Register file in the pipeline of the processor.

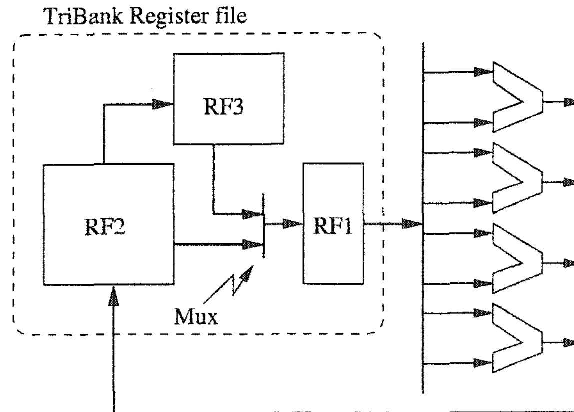


Figure 2.5 A TriBank Register file organization.

The bank RF1 consists of a small number of registers and a sufficient number of ports to support the issue width of the processor. The functional units are always supplied with the data only from the registers in RF1.

The bank RF2 consists of a large number of physical registers and a few read and write ports. The physical registers in RF2 are used for logical to physical register mapping at dispatch stage. Results are always written to registers in RF2.

The bank RF3 also consists of a large number of registers and a few ports. Registers in RF3 hold the values until the time they are freed when the conditions for precise exception handling are met.

From Figures 2.2 and 2.3, we observe that the average time during intervals t2-t0 (= S0+S1) and t4-t3 (S3) is around 45-50% each. Thus, A register value will spend duration t2-t0 (= S0+S1) in RF2 bank, and duration t4-t3 (S3) in RF3 bank. The bank RF2 holds the register value typically during the first half of lifetime of its mapping, while the bank RF3 holds the value for the second half of the lifetime. Therefore, we propose that there be an equal number of registers in RF2 and RF3 banks.

The RF1 bank only obtains those register values, from RF2 or RF3, that are soon to be consumed, and holds the values in the active state S2. The register values in RF2 are written to RF3 whenever RF3 has free registers and thus simultaneously freeing the corresponding registers in RF2. Section 2.3.2 discusses in detail the process of

transferring register values from RF2 to RF3, and conditions for freeing of registers in RF2 and RF3. Thus, the register in RF2 is freed up much earlier than that is done in a conventional monolithic register file. This leverages the processor to process a larger number of in-flight instructions to draw higher ILP.

The following subsection discusses in detail the various mechanisms for register value transferring to RF1 for consumption.

2.3.1 Register Value Fetching to RF1

The issue stage in the processor consists of wakeup logic and select logic. The instruction is said to be in wakeup state when it is in the reservation station waiting for both its source operands to be ready. When an instruction has all its source operands ready, it sends a ready signal to the select logic. The select logic sends a grant signal to the instruction permitting it to be executed, when necessary functional unit is available. The writing of register values to RF1 from RF2 or RF3 occurs when the values are ready to be consumed by an instruction that is ready to be issued for execution. To avoid any delay, the source operands for an instruction are fetched to RF1, when the instruction in the wakeup logic sends a ready signal to the select logic. This mechanism ensures that the values are fetched to RF1 just in time to be consumed, and also not much in advance before being consumed.

It is necessary that all active registers to be consumed soon be kept in RF1. Thus we proposed that RF1 be maintained as a small fully associative register file, similar to the Multi-banked register files proposed by Cruz *et al* [15]. In this case, the values are replaced according to the least recently consumed (LRC) policy. The registers in RF1 can also be easily marked as consumed or not-consumed, and replace only those registers that are flagged. A pitfall with this scheme is that the register value, though required, may not actually be read from the register file, but is supplied directly via the bypass logic. In that case, the register would not be marked as consumed and hence never replaced. To avoid this, instead of setting the flag for a register when it is read, the flags are set whenever the consumer instruction is executed, irrespective of whether

the source operands are read from RF1 or obtained via the bypass logic.

At the microarchitecture level, it is easy to identify when the register is first consumed (in fact nearly 85% of time, for both integer and floating point applications, a register value is consumed at most once). However, it is not possible to know in advance when it is consumed for the last time without a large overhead of keeping track of all the instructions that are customers for the operand. Hence, retaining the value in RF1 until it is completely consumed cannot be guaranteed with 100% accuracy. However, the register value replacement in RF1 using LRC policy increases the chances that the value is held for long enough to be consumed more than once if needed. Even with a rare chance that the value is replaced before it is consumed for the last time, it can be recovered from RF2 or RF3, wherever the register value is currently held. This can be achieved in a simple way as follows. Every time a source operand for an instruction that is ready to be issued is being written to RF1, a check is performed if the operand is already present in RF1. If the operand is not present, the value is written into RF1 from RF2 or RF3.

The above mechanism also helps in recovering from branch mis-predictions as follows. In a conventional architecture, when a branch mis-prediction occurs, the instructions that are not yet committed following the mis-predicted branch are squashed from the pipeline along with the corresponding values written in register file and the logical to physical register mappings due to those instructions. Subsequently, for instructions that are issued the source operands are read according to the logical to physical register mapping performed before the branch instruction. In the proposed architecture, the corresponding values existing in RF2 and RF3 are squashed when branch mis-prediction occurs. For new instructions issued, the source operand values are written to RF1 from either RF2 or RF3, wherever they exist. Consider the example shown in Figure 2.6.

Initially, the logical register lr6 is mapped to physical register pr3, and thus is read as a source operand for the next instruction. When the branch is predicted to be not taken and the following instructions are fetched, the logical register lr6 is renamed to a physical register pr2, different from the earlier mapping. The subsequent dependent

Code with logical registers	Code with renamed registers
lr6 ← ... ;	pr3 ← ...
... ← lr6 ;	... ← pr3
branch to LOOP ;	branch to LOOP
lr5 ← lr4 ;	pr8 ← pr9
lr6 ← ... ;	pr2 ← ...
... ← lr6 ;	... ← pr2
.... ;	....
LOOP: ... ← lr6 ;	... ← pr3

Figure 2.6 Example code 1.

instructions read from physical register pr2. However, when the branch is realized to be mis-predicted, the instruction in the correct path after the branch requires the logical value from lr6 which actually refers to pr3. Thus when the recovery mechanisms are initiated and the instructions are processed on the correct program path, the instruction requiring the value in physical register pr3 will find the register value in RF3 bank. This happens as the register value in pr3 has already been moved from RF2 to RF3 when the instruction preceding the branch was executed. If the second instruction in the above shown code does not exist, the mechanism remains the same except that the instruction at *LOOP* has to obtain the value to RF1 from RF2. This happens as the register value in pr3 is never consumed so far, and hence is still in RF2.

Overall, the mechanism of checking for values in RF1 before writing from RF2 or RF3 ensures the following:

* When the operand is to be consumed the first time, it is always transferred from RF2 to RF1.

* When the operand is to be consumed the second time and beyond, it is either already present in RF1, or is always transferred to from RF2 or RF3.

* When a branch mis-prediction occurs, the instructions issued after the pipeline are squashed, and the subsequent instructions obtain their source operands to RF1 from either RF2 or RF3 depending on the state of the instructions that precede

the mis-predicted branch.

2.3.2 Freeing of Registers in RF2 and RF3

A register in RF3 is freed according to the conditions followed in the case of a conventional monolithic register file. That is, for a current logical to physical register mapping, the physical register is freed when a subsequent instruction with same logical destination commits. The freed register is entered into the pool of free registers at dispatch stage. In our architecture, when the conditions of freeing a register are satisfied, the corresponding register in RF3 is freed up and can assume another register from RF2. When the corresponding entry in RF2 is written to the free register in RF3, the corresponding register in RF2 is freed and is entered into the pool of free registers at dispatch stage. The mapping of registers between RF2 and RF3 can be done in a direct-mapping (one-to-one) fashion.

In a direct-mapping (one-to-one) scheme, the two register banks RF2 and RF3 are maintained to be of same size, and the fetching of values from RF2 to RF3 is done strictly in a one-to-one correspondence. Whenever a register in RF3 is freed, the corresponding entry in RF2 is written to that free register in RF3 and the RF2 register is freed. From the processor's point of view in a conventional monolithic register file, the logical to physical register mapping lifetime is between the time physical register is allocated (t0) and the time when it is freed for next allocation (t4). In the direct-mapping policy, for applications in which during the lifetime of a logical to physical mapping, if the latency of quiescent state S3 for a register is much larger compared to the latency in state S0+S1, most of the time the register in RF2 will be written to RF3 sometime after it is consumed. In this case, the effective lifetime of a register mapping in the processor's view is the latency in freeing the register in RF2, which translates to the long latency of freeing a register from RF3. However, if the latency in state S0+S1 is larger than that in state S3, a register value might be written from RF2 to RF3 even before it is consumed. This will not hinder the reading of operands as the value can still be fetched to RF1 from RF3. However, there is a pitfall in this scenario. Consider a case when a

value in physical register pr5 is moved from RF2 to RF3 even before it is consumed and the register pr5 in RF2 is freed for next allocation. Subsequently the register pr5 obtains another value corresponding to the next logical renaming. A following instruction that sources the logical operand corresponding to pr5 in RF3 (former mapping) or the logical operand corresponding to pr5 in RF2 (later mapping) has to read the value correctly. To illustrate the scenario more clearly, consider the example shown in Figure 2.7.

```
Code with logical registers    Code with renamed registers
lr6 ← ... ;                    pr5 ← ...
lr2 ← ... ;                    pr9 ← ...
............ ;                 ............
............ ;                 ............
lr4 ← lr2 ;                    pr5 ← pr9
............ ;                 ............
... ← lr6 ;                    ... ← pr5 (in RF3)
... ← lr4 ;                    ... ← pr5 (in RF2)
............ ;                 ............
```

Figure 2.7 Example code 2.

Sometime in between the first two instructions shown, the value in pr5 is moved from RF2 to RF3, because previous pr5 in RF3 got freed up. Thus pr5 in RF2 is free to be allocated and suppose gets mapped to lr4. For the next two instructions shown, the reading of operands lr6 and lr4 happens to be from pr5 and thus operands have to be read as per the correct mapping. This scenario can be addressed by the following mechanism. When the value in pr5 is moved from RF2 to RF3, the register mapping table in the dispatch stage is updated by setting a flag for the mapping lr6==pr5, along with for all those instructions sourcing lr6 and are beyond dispatch stage in the processor pipeline. Thus any subsequent instructions that source lr6 will be informed accordingly to read the value from pr5 in RF3, and instructions that source lr4 will be indicated to read the value form pr5 in RF2. At some time later, when pr5 in RF3 is released and the value in pr5 in RF2 (corresponding to lr4) moves to RF3, the flag is set for the mapping lr4==pr5. This always ensures the reading of the operands from the correct register bank.

2.3.3 Impact on Bypass Logic

The design of a register file, apart from having an impact on the cycle time and the memory bandwidth available for the processor, also impacts the complexity of the bypass logic. A conventional monolithic register file with one cycle latency will have one level of bypass. However a large monolithic register file to support an 8-issue superscalar processor (requires around 128 registers with 16 read and 8 write ports) is unlikely to be implemented with one-cycle latency. Subsequently, a register file with a two-cycle latency has a negative impact on the processor performance due to increased branch mis-prediction latency [73]. Also, it requires two levels of bypass logic which incurs a significant cost. For a register file with two-cycle latency, designing only one level of bypass logic further degrades the performance [15]. For the proposed architecture, the complexity of the bypass logic is not affected and is the same as for a register file with single-cycle latency and a single level of bypass logic. This is due to the fact that the functional units are always supplied with the operands only from the RF1 register bank.

2.3.4 Handling Precise Exceptions

Instructions are fetched and retired in order while they are issued out of order. The retire mechanism assigns each mapped instruction a slot in a circular in-flight window (in fetch order). After an instruction starts executing, it can retire whenever all previous instructions have retired and it is guaranteed to generate no exceptions. The retiring of an instruction makes the instruction non-speculative guaranteeing that the instruction's effects will be visible to the programmer. The Alpha 21264 processor [39] implements a precise exception model using in-order retiring. The programmer does not see the effects of a younger instruction if an older instruction causes an exception. The retire mechanism also tracks the internal register usage for all in-flight instructions. Each entry in the mechanism contains storage indicating the internal register that held the old contents of the destination register for the corresponding instruction. This (stale) register can be freed for other use after the instruction retires. After retiring, the old destination register value cannot possibly be needed. An exception causes all younger

instructions in the in-flight window to be squashed. These instructions are removed from all queues in the system. The register map is backed up to the state before the last squashed instruction using the saved map state. The map state for each in-flight instruction is maintained, so it is easily restored. The registers allocated by the squashed instructions become immediately available.

In the Tribank register file organization, the speculative values can exist in any of the three register banks. A register map table is maintained with the status of each of the register in three banks, as against the maintenance of a register map table for a conventional monolithic register file. When an exception occurs, all younger instructions in the in-flight window will be squashed. These instructions are removed from all queues in the system, and the register map state is rolled back to the state before the exception causing instruction. The corresponding register values allocated to be consumed from RF1 are squashed. The registers in RF2 allocated by the squashed instructions are immediately available for logical register mapping, and the registers in RF3 holding the values by squashed instructions are made available for the transfer of values from corresponding registers in RF2. If registers in RF2 and RF3 with the same entry are cleared, the register in RF2 is soon allocated with a new mapping at dispatch stage. When the register in RF2 obtains the value, only then is the entry transferred to the corresponding free register in RF3. Until then the register in RF3 remains vacant.

2.4 Performance Evaluation

2.4.1 Simulation Methodology

We used Simplescalar-3.0 [9] for the Alpha AXP instruction set to simulate a dynamically scheduled out-of-order issue superscalar processor with the simulation parameters summarized in Table 3.1, with a few modifications as below. In Simplescalar the instruction issue queues and the re-order buffer (ROB) constitute one single centralized circular structure called the Register Update Unit (RUU). The simulator has been modified to model the issue queues that are smaller than the ROB size. Besides, an Alpha 21264

Table 2.2 Configurations for various register file organizations simulated. A bus from RF2 to RF3 indicates additional one read port for RF2 and one write port for RF3. Access time is measured at 0.18μ.

Index	*Configuration (IW = Issue Width)* read port (rp) , write port(wp), access latency	$(IW) = 4$ *access time (ns)*	$(IW) = 8$ *access time (ns)*
C1: conventional	RF = 128 registers, rp = 2*IW , wp = IW	1.0614	1.4873
C2: two-level organization	RF1 = 16 registers, rp = 2*IW , wp = IW, 1 cycle RF2 = 128 registers, rp = IW , wp = IW, 2 cycles	0.8046 0.9428	0.9791 1.2302
C3: TriBank organization	RF1 = 16 registers, rp = 2*IW , wp = IW, 1 cycle RF2 = 64 registers, rp = IW , wp = IW, 1 cycle RF3 = 64 registers, rp = IW, 1 cycle number of buses from RF2 to RF3 = IW	0.8046 0.8922 0.8687	0.9791 1.1552 1.0844
C4: TriBank organization	RF1 = 16 registers, rp = 2*IW , wp = IW, 1 cycle RF2 = 128 registers, rp = IW , wp = IW, 2 cycles RF3 = 128 registers, rp = IW, 2 cycles number of buses from RF2 to RF3 = IW	0.8046 1.0614 0.9428	0.9791 1.4873 1.2302

processor [39] based architecture is implemented with split integer and floating-point physical register files and issue queues for a 4-wide and an 8-wide out-of-order issue processor. The configurations for four different register file organizations are used for the analysis as shown in Table 2.2. To focus the analysis on the performance of various register file organizations, and not introduce an additional bottleneck in instruction processing, the architectures are simulated with sizes of 128, 256, and 256 for issue queue, ROB and LSQ, respectively. The benchmark programs are simulated for 500-1000 million instructions depending on the characteristics of each program, and the simulation was fast-forwarded past the initial warm-up phases.

The configuration C1 is a base processor implementation. In C1 the monolithic register file is implemented as a single cycle register file with one-level bypass logic.

The configuration C2 is implemented in line with the two-level register file design proposed by Cruz *et al* [15].

The configuration C3 is used to evaluate the performance of the TriBank register file organization with RF2 and RF3 constituting 64 physical registers as against a conventional monolithic register file with 128 physical registers. This constitutes an even-handed comparison of the TriBank scheme with C1 and C2 in terms of number of

physical registers available for data storage. However, note that the physical register bandwidth available for logical register mapping in C3 will be half of that available in case of C1 and C2.

Subsequently, to measure the performance of TriBank scheme with same logical to physical register mapping bandwidth, we also evaluate the configuration C4. In C2, C3 and C4, the small register bank closer to ALU is implemented as a single cycle one-level bypass register file. The RF2 and RF3 in C3 are implemented with a single cycle latency, while RF2 in C2, and RF2 and RF3 in C4 are implemented with a two-cycle latency.

We use SPEC2000 benchmarks, and evaluate both the integer and floating-point programs. We used the access time models of CACTI-2.0 [80] at 0.18μ technology, with necessary modifications to generate cycle times for multiported register files, to evaluate the complexity of proposed register file structures in comparison to the baseline organization. The use of CACTI-2.0 was greatly expanded, since the tool is made to analyze caches with few ports. Here we have made use of it for register files (which typically do not use sense amps like caches) with a larger number of ports. We compute the access time of the RF1 register bank while accounting for associativity.

2.4.2 Results and Analysis

The results obtained with the simulation of various register file configurations in a 4-wide and an 8-wide out-of-order issue superscalar processors are shown in Figures 2.8 to 2.13.

Figures 2.8 and 2.9 show the IPC throughput for various integer and floating-point Spec2000 programs for 4-wide and 8-wide processors, respectively. The degradation in IPC for configuration C2 as compared to configuration C1 is in line with the analysis given by Cruz *et al* [15]. It is observed, that the TriBank register file configurations C3 and C4 perform either similar or better as compared to the base configuration. An enhancement in IPC by 2% and 3% for a 4-wide processor, and by 2% and 1% for an 8-wide processor, is seen with configuration C3, for SpecInt and SpecFP programs, respectively. On the other hand Implementation of configuration C4 enhances IPC by

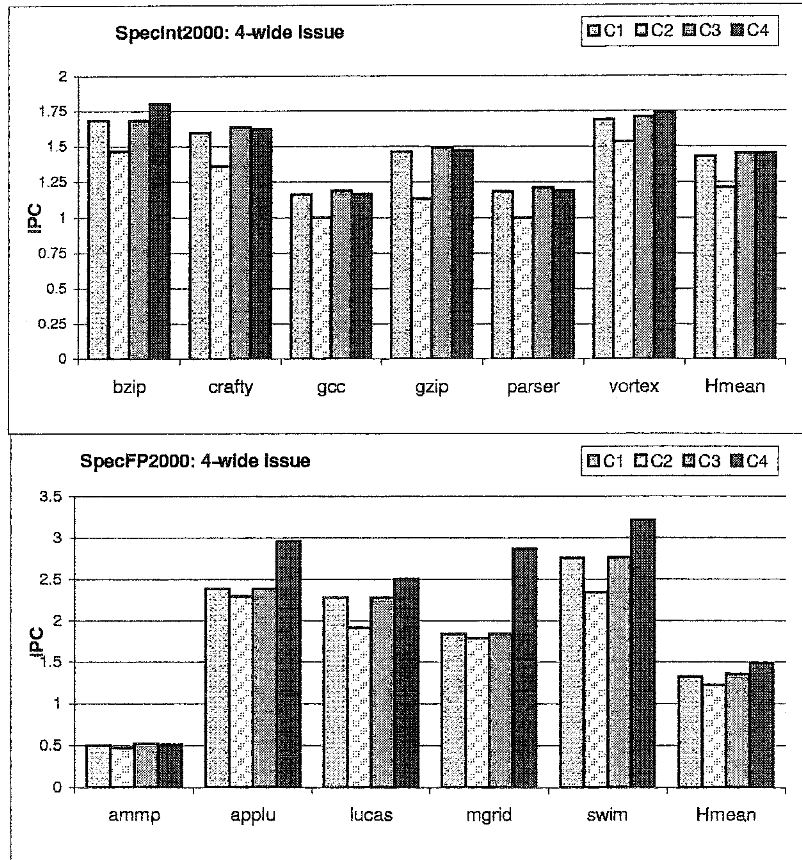


Figure 2.8 Instructions per cycle (IPC) throughput for various register file configurations in the 4-wide issue processor.

2% and 12% for a 4-wide processor, and by 3% and 14% for an 8-wide processor for SpecInt and SpecFP programs, respectively. For certain integer benchmarks like crafty, gcc, gzip, and parser, C3 is observed to be performing slightly better than C4. This is due to the larger access cycles for register banks RF2 and RF3 in C4 as compared to those in C3. Hence, this performance difference can vary in either way depending on the architecture implementation technology and the other factors that govern the access time of a register bank.

Figures 2.10 and 2.11 show the relative instruction throughput of the processor per second, for 4-wide and 8-wide processors, respectively, when the register access time is factored in. This performance measurement is done assuming that the register file

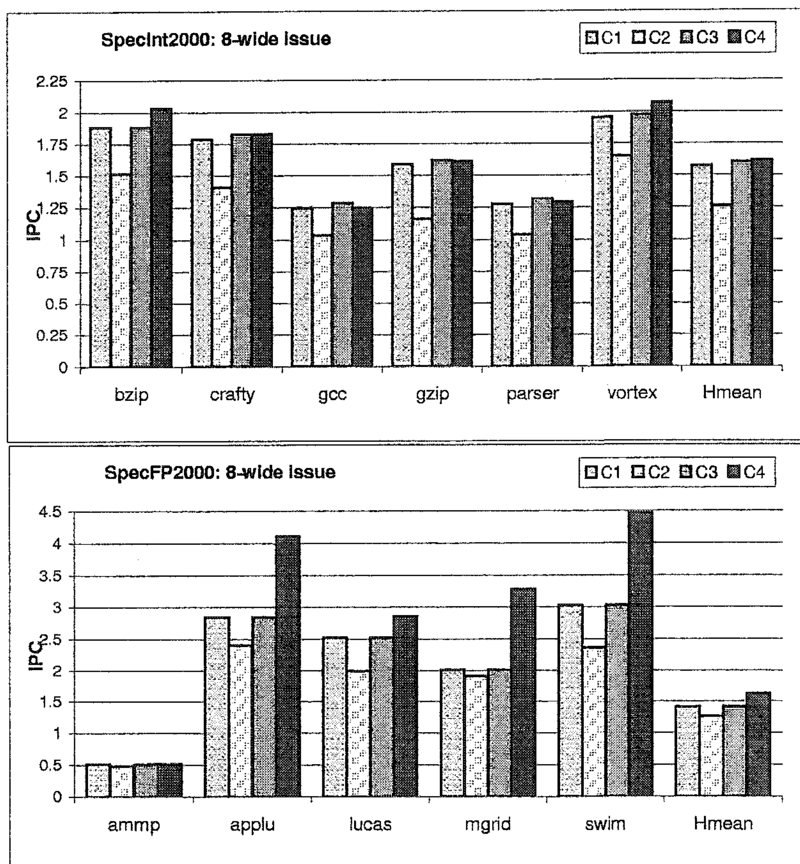


Figure 2.9 Instructions per cycle (IPC) throughput for various register file configurations in the 8-wide issue processor.

access time determines the cycle time of the processor. An enhancement in instruction throughput by 34% and 35% for a 4-wide processor, and by 54% and 52% for an 8-wide processor, is noted with configuration C3 for SpecInt and SpecFP programs, respectively. On the other hand, implementation of configuration C4 enhances the instruction throughput by 35% and 57% for a 4-wide processor, and by 56% and 96% for an 8-wide processor.

The advantage gained by the inclusion of RF3 register bank, used to retain the register values before being freed, is explained as follows. Cruz *et al* [15] have shown that a large RF2 (with large access time) and small RF1 results in IPC loss though instruction throughput per second is gained as it increases the pipeline latency. We

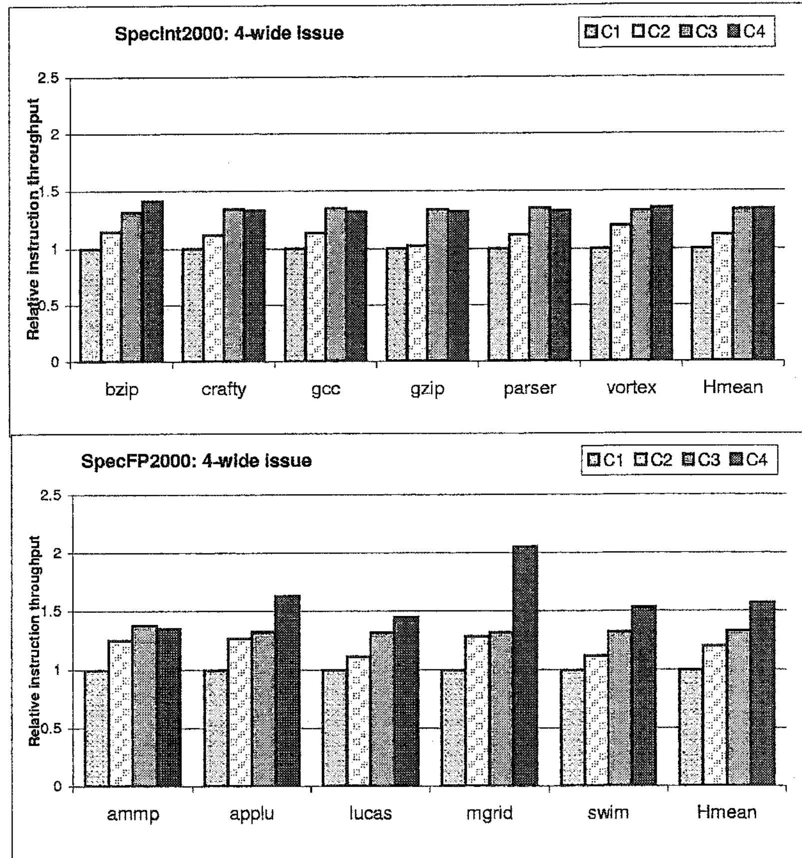


Figure 2.10 Relative instruction throughput when register access time is factored in for various register file configurations in the 4-wide issue processor.

have shown that splitting their large register bank into RF2 and RF3 provides the same memory bandwidth for dispatch stage while reducing pipeline latency improving both IPC and the instruction throughput. This is a significant contribution.

Figures 2.12 and 2.13 illustrate *dispatch factor*, the percentage of runtime when at least one free physical register is available for logical register mapping. It is noteworthy that for architectures provided with large instruction processing structures (issue queues, ROB, and LSQ), only a very large register file would be able to obtain a dispatch factor of 100%, i.e., a free physical register is available for mapping all through the run time. The implementation of configuration C4, wherein a register bank is viewed by the processor for logical to physical mapping and another register bank is provided to

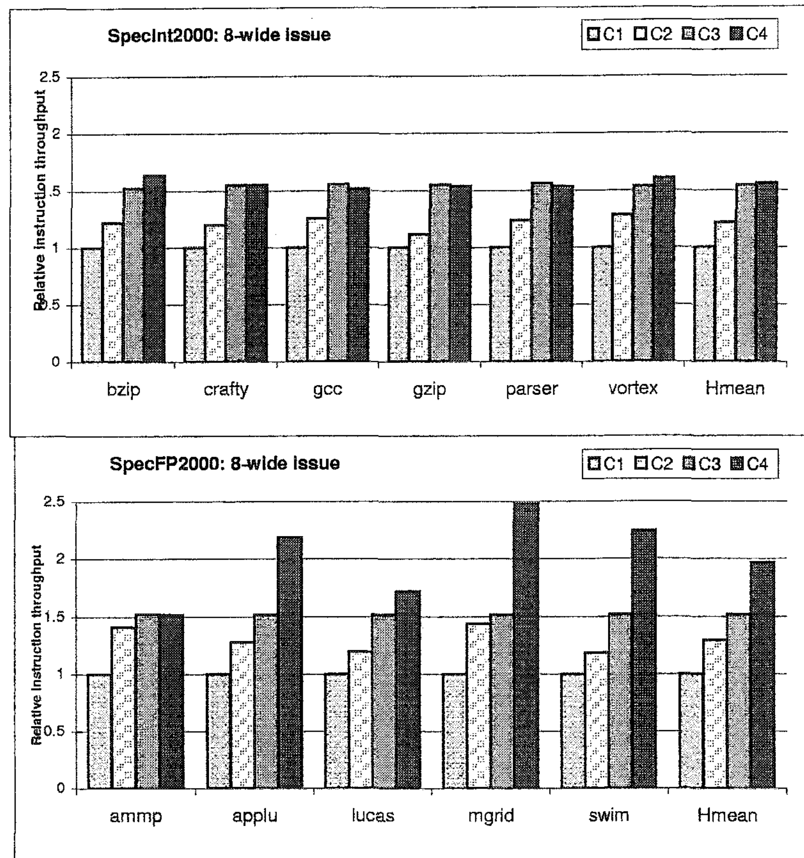


Figure 2.11 Relative instruction throughput when register access time is factored in for various register file configurations in the 8-wide issue processor.

hold the values during the latter part of their lifetime until they are freed, enhances the overall performance of the processor. Also, this design performs better, as each of the two register banks will have a smaller access time as compared to a single large register bank designed to provide the large memory bandwidth.

2.5 Related Research

An extensive and widespread research has been conducted to bridge the increasing processor-memory performance gap. A lot of research has focused on reducing or tolerating the large memory access latencies, and innumerable techniques have been de-

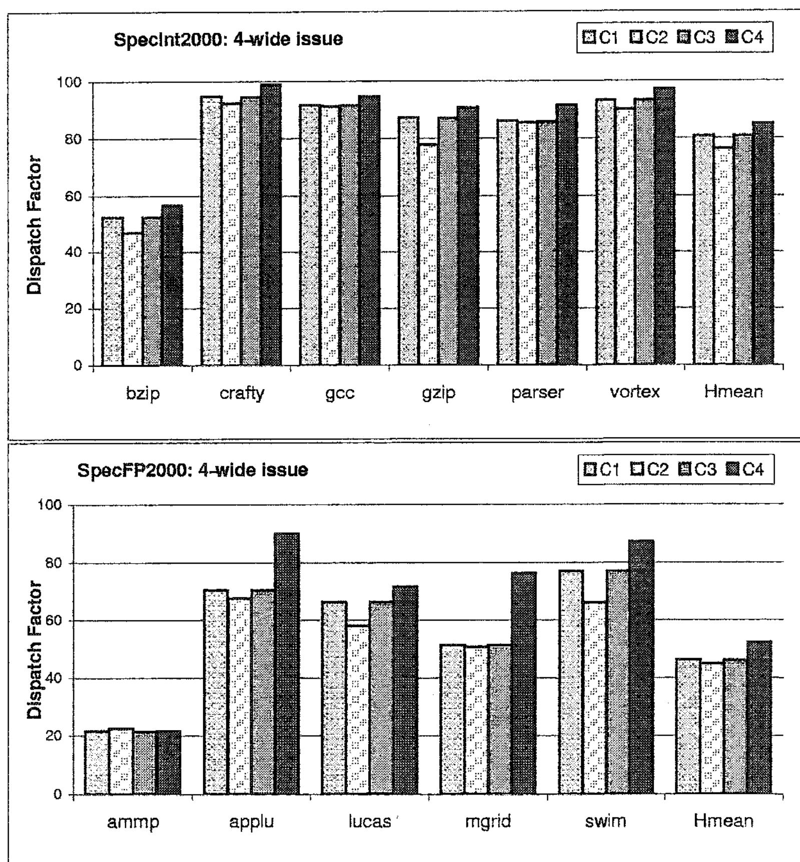


Figure 2.12 Percentage of run time during which at least one free register exists for various register file configurations in the 4-wide issue processor.

veloped in that direction. To review all of them here is an arduous task. Burger *et al* [10] assert that aggressive implementations of latency-tolerance techniques in future processors will expose memory bandwidth as a severe bottleneck. While proposing a number of solutions, they hypothesize that all system memory will eventually be coupled with the processor on the die, enabling performances much higher than the current ones. Farkas et al [24] investigated and analyzed the performance trends and design relationships among the register file and the other levels of data memory hierarchy. Chen and Somani [13] presented a methodology to assess the performance trade-offs of various architecture techniques based on the equivalence of mean memory delay time.

Various techniques have been proposed to enhance the register access time while

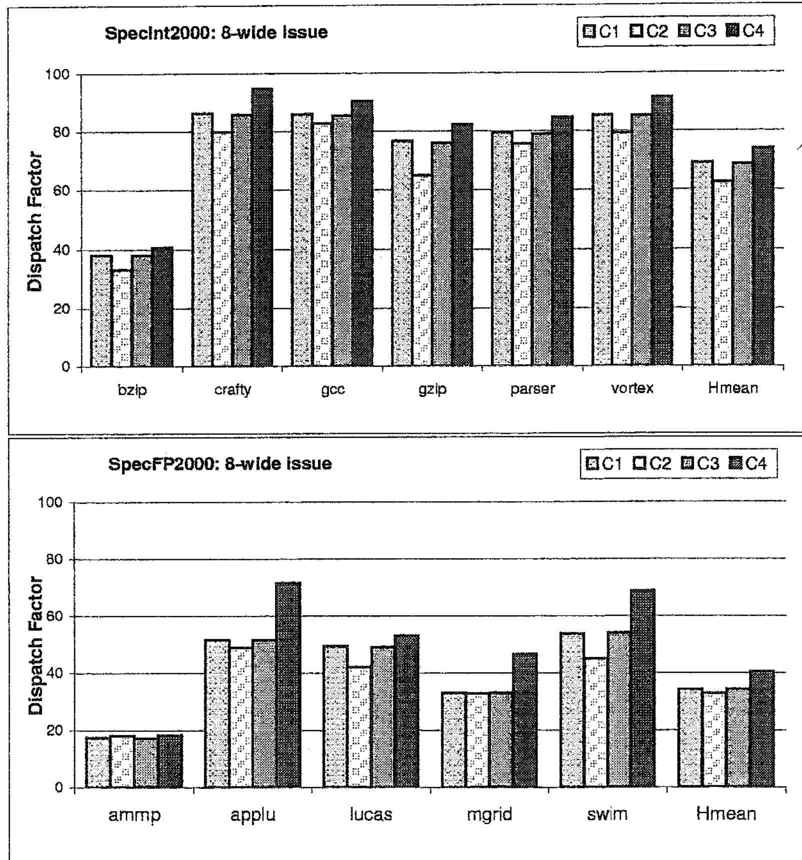


Figure 2.13 Percentage of run time during which at least one free register exists for various register file configurations in the 8-wide issue processor.

simultaneously striving to provide a large register file. The Alpha 21264 microprocessor [39] uses a replicated register file organization to reduce the number of ports, wherein each copy can be accessed by only a few functional units. Reducing the number of ports is an effective technique to reduce register access time. Recently, Tseng *et al* [72] have examined the designs of such multiple bank with fewer ports to reduce power and area. However, the access time is still a problem as long as the demand for larger register files grows.

Cruz *et al* [15] used a multiple-banked organization for implementing a two-level register file. The level one (L1) smaller register file maintains a subset of registers in level two (L2) large register file and accommodates those registers that are active

suppliers of values to the functional units. Schemes for transferring the register values from L2 to L1 are proposed. This organization takes advantage of the latency in the quiescent states S0 and S1 when the register is waiting to be consumed and the small active state S2. The register value is fetched from L2 to L1 when ready to be consumed. The organization performs well in reducing the register access time by maintaining a small L1 register file. However, the scheme does not exploit the long latency in quiescent state S3, as the L2 register file still holds the values until the registers are freed. The L2 register file is used for logical to physical register mapping. Hence, this does not support the architectures that use aggressive techniques for processing a large number of in-flight instructions to draw higher ILP, where availability of free physical registers becomes a bottleneck at the dispatch stage. Further, implementation of a large L2 register file results in a large latency in access from L2 to L1.

Balasubramonian *et al* [2] proposed and evaluated two orthogonal designs - two level and multi-banked register file. In the two-level organization, L1 structure is used for logical to physical register mapping and holds the values until they are consumed. After the consumption the register is moved to L2 and maintained until it is freed. This organization exploits the quiescent state S3 where the register is waiting to be freed. Such implementation helps in organizing a relatively smaller L1 register file as compared to a conventional single monolithic register file, and hence reduces the register access time. However, reducing the number of registers in L1 which is used for register mapping increases the possibility of non-availability of free registers for mapping, and hence results in a more number of stalls at the dispatch stage. The approach in this case and in [34], with multiple interleaved register banks, results in difficulty in managing the complexity and the additional latency of the control logic required to handle read and write bank conflicts and the mapping of register ports to functional units.

Our *TriBank register file* architecture is motivated from the two architectures in [15] and [2]. Reducing the register access time even while maintaining a steady availability of free registers for mapping is essential for higher performance in dynamic superscalar processors, and our scheme strives to achieve that by taking advantage of the results

and conclusions drawn from these designs.

Apart from the above, various other techniques have been proposed in the past for an effective utilization of register resources. Borch *et al* [4] have recently proposed the caching of registers. A software-controlled two-level hierarchical register file organization was implemented in Cray-1 [62]. Swensen and Patt [68] proposed a hierarchical non-inclusive register file for a statically scheduled architecture, where register allocation is performed by the compiler. Wallace *et al* [78] proposed a scalable register file architecture, that uses multiple banked register file and maps the result to a physical register at the write stage in the processor pipeline. Llosa *et al* [49] proposed a *non-consistent dual register files*, where local register instances are stored in the local register file of the cluster, while register instances used by both clusters are replicated on both clusters. Yung *et al* [86] proposed a *Register Scoreboard and Cache* scheme wherein a subset of the registers are cached in a fast bank with an LRU replacement policy. For media processing, Rixner *et al* [60] presented a taxonomy of partitioned register file architectures across three axes - data parallel, instruction-level parallel, and memory hierarchy. The register deallocation policies have been discussed in detail by Moudgill *et al* [52], where they proposed a mechanism that implements register renaming, dynamic speculation and precise interrupts.

The HP PA-8000 [43] processor implementation maintains a logical register file that holds committed values, and the rename registers are maintained in a separate buffer. To reduce the register file access time, Tremblay *et al* [71] proposed a *3-D Register File*. In that, it was shown that the data array portion of the register file can be significantly reduced by designing the register file in multiple planes, where a plane in 3-D is a set of registers in 2-D. A cluster based design of execution units and the extension of storage hierarchy for each cluster, in place of a global register file, is introduced by Dally [16]. The work investigates the limitations of the dedicated-wires based global communication in the interconnection structures, and proposes alternative approaches of clustering, register hierarchy and global networks.

Zyuban and Kogge [87] have developed energy models for multi-ported register files

with a variety of architectural parameters, and assert that the centralized register files would become the dominating power component of next-generation superscalar computers. A *Split register file* architecture was proposed by them as an energy-efficient alternative design. Chang *et al* [12] have proposed a *conjugate register file* scheme to handle an aggressive instruction scheduling in superscalar processors. Gonzalez *et al* [27] proposed a *virtual registers* architecture with a strategy to reducing the pressure on register file by delaying the allocation of physical registers until instructions complete, instead of doing it in the decode stage.

2.6 Summary

Microprocessor researchers and designers are continuously facing the uphill task of bridging the increasing processor-memory performance gap, and also address the need for a larger memory bandwidth to meet the requirements of future wide-issue superscalar processors. In the process, the design of an effective register file architecture is becoming a bottleneck to the enhancement of the performance. The two main goals in designing an effective register file organization is to provide a small register access time to enable a faster processor cycle time, and provide a large number of registers to enable dispatching as many instructions as possible to issue window for extracting higher ILP. To meet these two goals simultaneously, we developed a *TriBank Register file* organization, a novel architecture that exploits the quiescent states in the lifetime of a logical to physical register mapping.

The TriBank Register file organization consists of three banks of physical registers with a heterogeneous structure. The Register file organization has been developed based on the observations that a physical register, in its one lifetime of a logical register mapping, first experiences a long quiescent state, then an active state for a short duration, and finally a quiescent state for a long duration before being freed up. Accordingly, each register bank is designed to consist a different number of registers and different number of ports according to the architecture requirements. Implementation of the TriBank register file organization, as compared to a conventional monolithic register file in an

8-wide out-of-order issue superscalar processor enhanced the throughput in instructions per cycle (IPC) by 3% and 14%. When the register file access time is factored in, the instruction throughput is enhanced up to 56% and 96%, for SpecInt2000 and SpecFP2000, respectively. The significant contribution of our proposed register file architecture is that it enhances IPC, when earlier work in designing effective register file has resulted in IPC degradation.

3 ADAPTIVE REGISTER FILE

3.1 Introduction

Based on the earlier observations, we propose *Adaptive Register file Architecture*, a novel architecture for dynamic superscalar processors that strives towards achieving greater performance by providing a larger memory bandwidth or higher on-chip computing power, depending on the requirements of the application. The processor is designed with a conventional register file and execution units. In addition, we design an *Adaptive Register file Computing* (ARC) unit, that dynamically switches roles as a register file or as a computing unit depending on the requirements of the task under execution. When an application demands higher computing bandwidth, the ARC unit performs the computations in parallel with the other available execution units. On the other hand, when an application demands larger memory bandwidth, the ARC unit provides an additional register bandwidth.

In the development of the architecture, we first identify and design a suitable compute intensive function to fit into the ARC unit. This implementation poses various challenges. The compute intensive function implemented in the ARC unit should be generic enough to be executed as a part of a wide range of applications, else the resources are under-utilized. Besides, the nature of the function needs to be in such a way that it can be broken down into multiple independent threads so that the ARC unit can perform the computations in parallel with the conventional execution units. Besides, the computation of function in the ARC unit in parallel with the conventional execution units should give considerable amount of speedup when compared to the computation of the function in the processor without the ARC unit. Further, this needs to be achieved

with a minimal hardware complexity such that the access time of registers in the ARC unit is less.

The rest of the chapter is organized as follows. In Section II, we review the related research. Section III presents the proposed architecture, and the design and implementation of a compute-intensive function in the ARC unit. Section IV presents the performance analysis of a wide-issue superscalar processor supplemented with an ARC unit in executing the compute-intensive function. Finally, Section V concludes the discussion.

3.2 Related Research

The first known attempts to use the memory elements for computation are that of by Kautz [38] and Stone [67]. During the past decade, attention has been drawn towards the significance of designing a reconfigurable coprocessor coupled with the general purpose processor [18]. Ye et al [85] developed *Chimaera*, a micro-architecture that integrates a reconfigurable functional unit (RFU) into the pipeline of a dynamically scheduled superscalar processor. A sequence of instructions is mapped for a single operation in the RFU unit, provided it reads up to 9 input registers and generates a single register output. The C compiler that automatically generates binaries for RFU operations is presented. Razdan *et al* [59] explored ways to incorporate hardware-programmable resources and described compilation/synthesis system that automatically exploits the resources to improve the performance of general purpose applications. The Garp architecture [11] combines reconfigurable hardware with a standard MIPS processor on the same die to exploit the better features of both. Kim *et al* [41] developed a *Reconfigurable Functional Cache* (RFC) based computing architectures, wherein a module of L1 data cache is used as a computing unit to execute a compute-intensive function in multimedia applications. In this design, a more coarse-grained functions, such as Discrete Cosine Transform (DCT), Finite Impulse Response (FIR), are mapped into the computing cache modules.

The register file architecture and its organization has been widely researched to enhance the memory bandwidth of wide-issue superscalar processors. However, this is

the first work, to the best of our knowledge, that proposes an adaptive register file organization that can enhance the on-chip memory bandwidth or provide higher on-chip computing bandwidth by acting as a computing unit, as the situation demands.

3.3 Adaptive Register File Architecture

The architecture consists of a main register bank that acts as a conventional register file that supplies the operand values to the functional units. Besides, the processor is embedded with an ARC unit, that is designed to act as an additional bank of registers, or a computing unit that processes matrix multiplication. The architecture with the processor embedded with an ARC unit is shown in Figure 3.1. The superscalar processor, with the specifications mentioned in Table 3.1, is used as the base architecture for the proposed design.

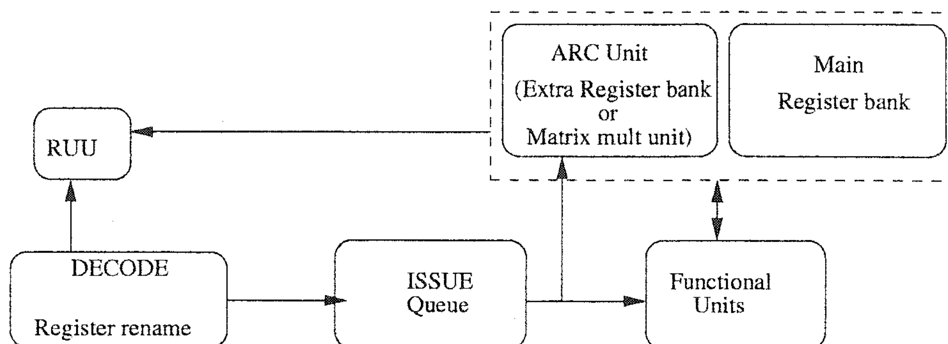


Figure 3.1 ARC unit placement in the processor pipeline.

When an application under execution demands a large memory bandwidth, the ARC unit is made to act as a register bank and hence provides an extra memory bandwidth. In a conventional superscalar processor, the logical destination register for an instruction fetched is mapped to a free physical register at the dispatch stage. Subsequent instructions with the same logical register as their source operand are assigned to read from the mapped physical register. The logical to physical register mapping remains active until another instruction with same logical register as its destination enters the

Table 3.1 Simplescalar simulation parameters.

Parameter	*Value*
Instruction cache *- latency*	32KB, 2-way, 32B line 1 cycle
Data cache *- latency*	32KB, 4-way, 32B line 1 cycle
Branch predictor *−mis − prediction latency* *−return address stack size*	bimodal, 2K table size 7 cycles 8
Instruction issue queue size	64 (INT and FP, each)
Load/store queue (LSQ) size	256
ReOrder buffer (ROB) size	256
Issue width	4 or 8
Commit width	4 or 8
Functional units *−Integer arithmetic* *−Integer multiplier* *−floating point arithmetic* *−floating point multiplier*	 8 4 4 4
L2 unified cache *−latency*	256KB, 4-way, 64B line 6 cycles
TLB *−D − TLB* *−I − TLB* *−latency*	 512KB, 128 entries 256KB, 64 entries 30 cycles
Memory *−latency first, next* *−bus width*	 70, 2 cycles 8B

dispatch stage. The logical destination of that instruction is then mapped to another free physical register and the process continues. The dispatch stage in the pipeline is stalled when no free physical register is available. The scheme of logical to physical register mapping eliminates the write after read (WAR) and write after write (WAW) data dependencies. The earlier allocated physical register is freed only when the subsequent instruction with same logical destination is committed. This is done to enable the recovery from branch mis-predictions and handle exceptions precisely. Thus, when a task demands a large memory bandwidth, ARC unit acting as an extra register bank provides larger number of physical registers to be mapped to the logical register at the dispatch stage. This enables larger number of instructions to be dispatched enabling the processor to view larger instruction window to draw higher instruction level parallelism. On the other hand, when an application demands higher computing bandwidth, the ARC unit performs the computations in parallel with the other available execution units.

3.3.1 ARC as a Computing Unit

The compute-intensive function chosen for implementation in the ARC unit should be generic and a widely used one in most applications. Else, the on-chip hardware resources allocated for computing such function will be under-utilized. Besides, the nature of the function needs to be in such a way that it can be broken down into multiple independent threads so that the ARC unit can perform the computations in parallel with the conventional execution units. Further, a design and implementation of such function should provide reasonable speedups as compared when the function is computed in a general purpose processor. Else, the overall speedup for the application, as per Amdahl's law, will not be significant enough to justify the overhead of on-chip hardware resources allocated for the purpose. The time to load the configuration (configuration overhead) prior to the computations in the ARC unit should be as minimal as possible to avoid the configuration overhead from becoming a bottleneck to the speedup that can be achieved. The number of memory elements (registers) used for the design of ARC unit,

the arrangement of those registers into various assemblies, the number of read and write ports, and the time to access a datum from such organization should be optimal to suit the requirements of a register file. For the above reasons, the design and implementation of the ARC as a computing unit is a challenging task.

In the current scenario where new and high performance-demanding applications are fast emerging, researchers and designers continuously experience a demand for significant increases in processor performance. Areas such as signal processing and imaging require enormous computing power, and thus more on-chip computing resources. A dissection of the algorithms used in these, and related applications, reveal that many of the fundamental actions involve matrix operations. Most of these operations are matrix multiplications, which are frequently occurring operations in a wide variety of real world algorithms. The Discrete Cosine Transform (DCT), the Discrete Fourier Transform (DFT), and Singular Values Decomposition (SVD), used in digital image/signal processing including compression and beam-forming applications are some of those applications [58, 45]. The multiplication of two matrices of size NxN each, in a general purpose processor consumes $\mathcal{O}(N^3)$ operations, requiring $\mathcal{O}(N^3)$ addition and $\mathcal{O}(N^3)$ integer multiplication operations (considering the elements in the matrix to be integers), and hence becomes a bottleneck to the performance of the processor. Therefore, larger number of on-chip processing elements that compute in parallel are required.

Subsequently, we choose the integer matrix multiplication function as a compute-intensive function to implement in the ARC unit. Consider two NxN matrices A = $[A_{ij}]$ and B = $[B_{ij}]$. The product C = $[C_{ij}]$ of the two matrices is given by

$$C = A * B \tag{3.1}$$

such that

$$C_{ij} = \sum_{k=0}^{N-1} A_{ik} B_{kj} \tag{3.2}$$

To reduce the complexity of the matrix multiplication operations, we design a 3-LUT based computation, wherein the time to configure the LUTs would be a maximum of eight clock cycles. The design of the ARC unit is shown in Figure 3.2. It consists of

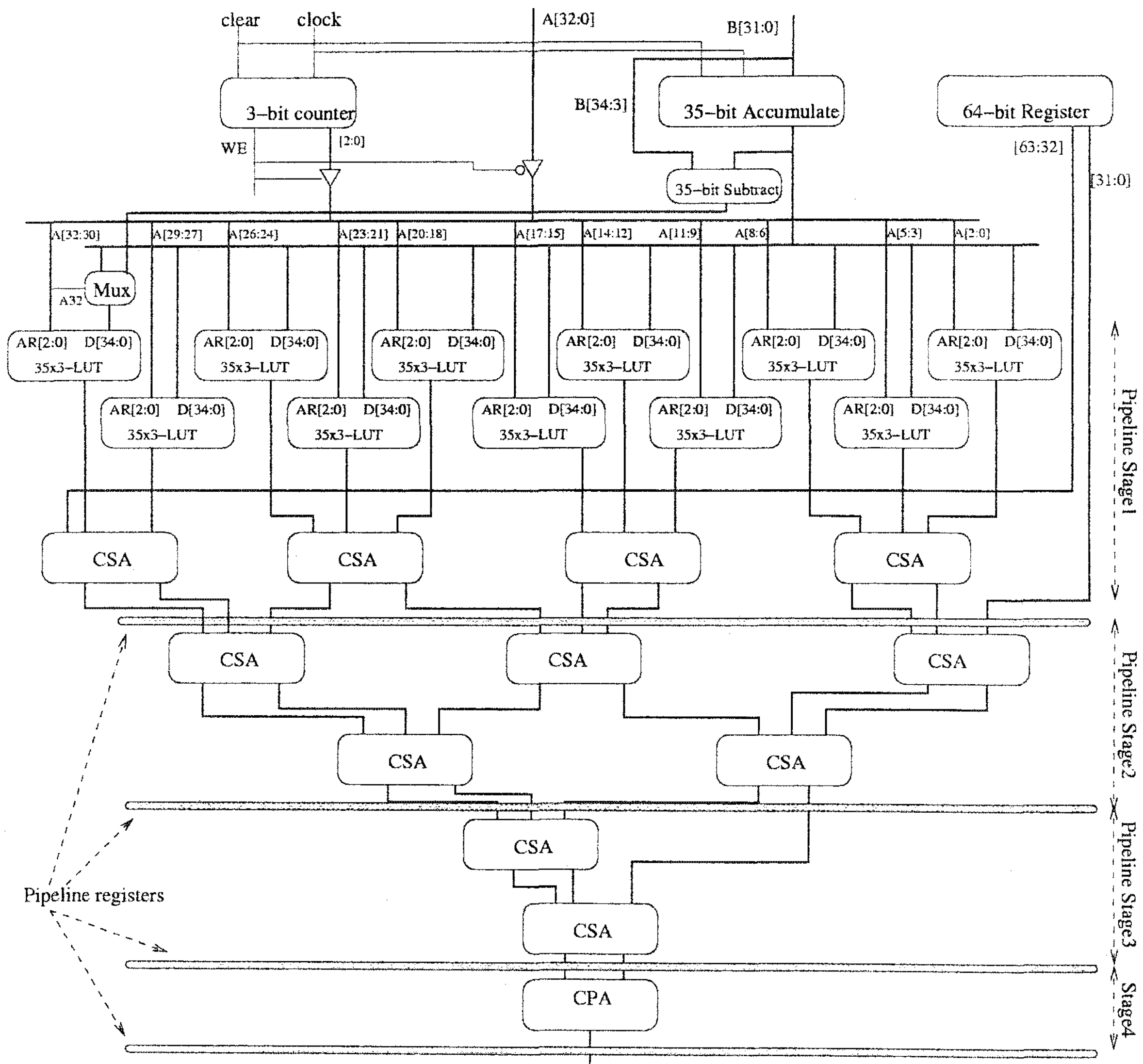


Figure 3.2 32-bit 8-cycle Reconfigurable Matrix Multiplier. The write enable (WE) and clock signals are connected (not shown in figure) to all the LUTs. The multi-stage addition of partial products using CSAs is pipelined as shown. The end result is computed in a carry propagate adder (CPA).

eleven 3-LUTs addressed by the 32-bit integer multiplicand (the sign bit is extended by one bit to make it a 33-bit integer). The addition of the partial products (PPs) obtained from the LUTs is performed by a multiple stage carry save adders (CSA). The matrix multiplication operation in the ARC unit consists of two stages - configuration and computation. For the purpose we introduce two new instructions - ARC-CONF with one operand A, and ARC-COMP with three operands C, B, and C (*dest*, *source*, *source*). The instruction (ARC-CONF A) when invoked, performs a write operation for eight clock cycles controlled by a 3-bit counter. The eight rows in each of the ten right-most 3-LUTs are loaded with the values of 0, A, $2A$, $3A$, $4A$, $5A$, $6A$, $7A$, respectively, as addressed by the output from the 3-bit counter. Simultaneously, the eight rows in the left-most 3-LUT are configured with 0, A, $2A$, $3A$, $-4A$, $-3A$, $-2A$, $-A$, respectively, to facilitate a signed integer multiplication operation. Since we assume a single LUT cannot be loaded in parallel, we partially load all the LUTs each cycle. Thus all the LUTs in the ARC unit are configured in parallel and hence the total configuration time for each element of matrix A is eight clock cycles. When the configuration of the LUTs is completed, the instruction (ARC-COMP C, B, C) is invoked. The operation consumes two source operands, wherein the first source operand is used to lookup the contents of the LUTs, while the second source operand is an intermediate result obtained from the earlier ARC-COMP operations. The eleven PPs obtained from the current lookup operation are combined with the two 32-bit parts of the intermediate result (it can be seen in Figure 3.2 that a proper bit-alignment is made to perform the addition of the thirteen partial products, i.e., the lower 32-bit value of the intermediate result is added to the least significant partial product, while the higher 32-bit value of the intermediate result is added to the most significant partial product). Subsequently, the addition of the thirteen PPs is performed in five stages of addition using CSAs and the result obtained thereafter is stored in the designated destination operand. The addition of PPs using CSAs and a carry propagate adder (CPA) is pipelined as shown in the Figure 3.2. The first stage constitutes the lookup operation and first level of CSA computation. Subsequently, four levels of CSA computations are placed in two stages (two CSA levels

in each stage). Due to the higher computation latency in a CPA as compared to a CSA, one CPA computation is placed in a single stage. The pseudo code for a multiplication of two NxN matrices and a sequence of instructions generated for the multiplication of two 2x2 matrices in the ARC unit is shown in Figure 3.3.

Algorithm 1. Matrix multiplication in ARC

for i=0 to (N-1)
  for j=0 to (N-1)
  ARC-CONF A_{ij}
    for k=0 to (N-1)
    ARC-COMP C_{ik}, B_{jk}, C_{ik}

for N=2, the sequence of operations for execution in ARC:
  Initialize $C_{00} = C_{10} = C_{01} = C_{11} = 0$;
  ARC-CONF A_{00}
  ARC-COMP C_{00}, B_{00}, C_{00}
  ARC-COMP C_{01}, B_{01}, C_{01}
  ARC-CONF A_{01}
  ARC-COMP C_{00}, B_{10}, C_{00}
  ARC-COMP C_{01}, B_{11}, C_{01}
  ARC-CONF A_{10}
  ARC-COMP C_{10}, B_{00}, C_{10}
  ARC-COMP C_{11}, B_{01}, C_{11}
  ARC-CONF A_{11}
  ARC-COMP C_{10}, B_{10}, C_{10}
  ARC-COMP C_{11}, B_{11}, C_{11}

Figure 3.3 Matrix multiplication in a ARC unit (shown excluding load/store and branch instructions).

Alternatively, the lookup operation can be performed using 4-LUTs. In that case, the addition of 10 PPs (eight from lookup and two from earlier intermediate result) still requires five stages of CSAs and hence results in same computation latency (in fact, the lookup time for a 4-LUT is slightly higher than that for the 3-LUT). However, the total number of memory elements (registers) placed in the ARC unit would be 128, as

compared to 88 in the 3-LUT based design. Thus, the 4-LUT based design results in a larger area overhead and with a slightly higher computation latency, and hence we prefer a 3-LUT based design for the implementation. Similarly, the lookup operations can be performed using 5-LUTs wherein the addition of nine PPs (seven from lookup and two from earlier intermediate result) can be performed in a four-stage CSAs. However, the lookup time for the 5-LUT is much larger and offsets the advantage of reduced stages in addition. Further, the total number of memory elements required for such operation is 224, which results in a very large area overhead on the chip.

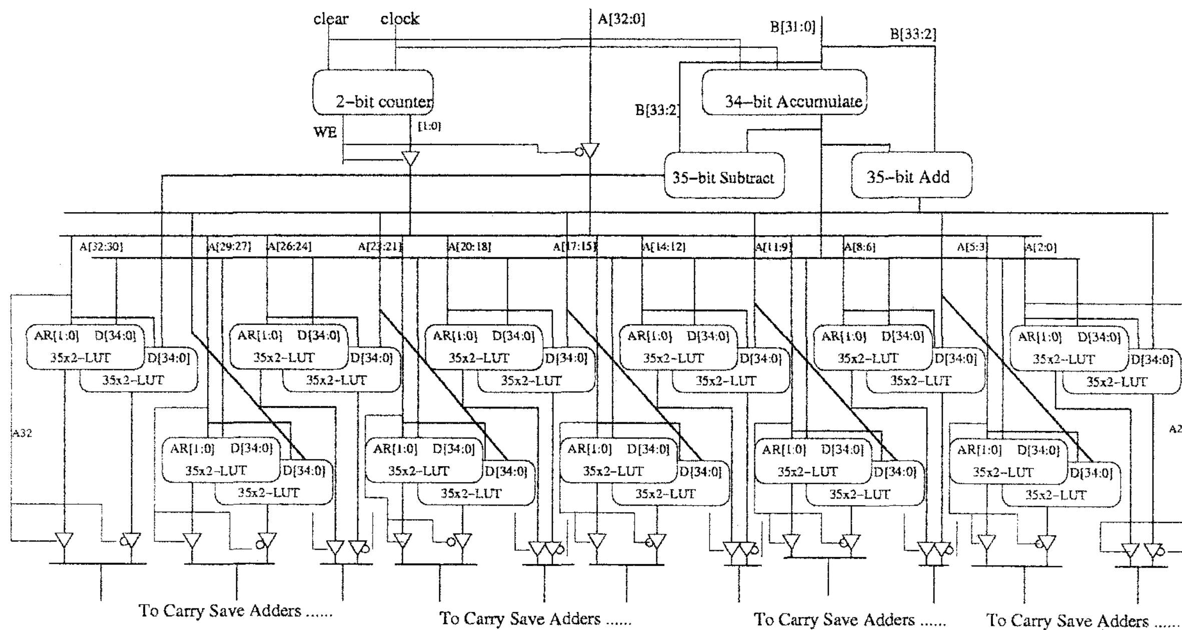


Figure 3.4 32-bit 4-cycle Reconfigurable Matrix Multiplier. The write enable (WE) and clock signals are connected (not shown in figure) to all the LUTs. The pipelined addition of partial products using CSAs is performed as shown in Figure 3.2.

The implementation of the ARC unit can further be modified to reduce the configuration time to 4 and 2 clock cycles, in line with the design of a *self configuring binary multiplier* proposed by Wojko and ElGindy [82]. The lookup operation with a configuration time of 4 cycles is shown in Figure 3.4. A 3-LUT is divided into two segments

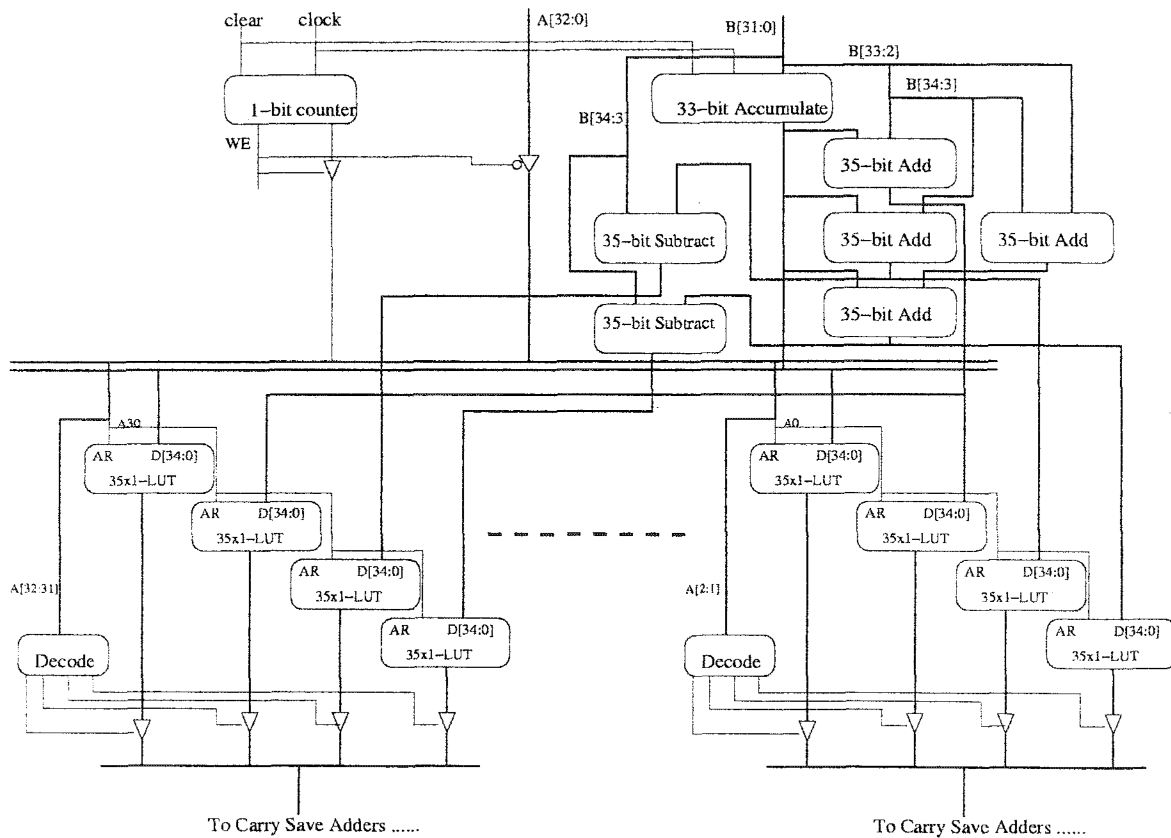


Figure 3.5 32-bit 2-cycle Reconfigurable Matrix Multiplier (partially shown). The write enable (WE) and clock signals are connected (not shown in figure) to all the LUTs. The pipelined addition of partial products using CSAs is performed as shown in Figure 3.2.

and the two segments of each 3-LUT are loaded with the respective contents in parallel. This is achieved with an additional 35-bit adder that adds a value of $4A$ to each of the consecutive outputs from the 34-bit accumulate. Thus it takes 4 clock cycles to load the values of 0, A, $2A$, and $3A$ into the rows in first segment, and simultaneously write the values of $4A$, $5A$, $6A$, and $7A$ into the rows of second segment. Among the three bits to perform the lookup in a 3-LUT, the two least significant bits (LSBs) are used for address lookup while the most significant bit (MSB) acts as a select signal for the multiplexer to choose either of the outputs from the two segments. Note that this is similar to a design where 16 2-LUTs are used for the lookup operation (with a total of 64 memory elements). However, in the 2-LUT implementation, the addition of 18 PPs (16 from lookup and 2 from earlier intermediate result) requires six stages of CSAs and hence results in a larger computation latency.

The lookup operation with a configuration time of 2 cycles is performed with the further segmentation of the 3-LUT, however with a larger area overhead due to additional adders and decoders required. The partial implementation of the design is shown in Figure 3.5. The total on-chip area consumed by the ARC unit, designed according to the above described three different configuration schemes and implemented at 0.18μ technology, is shown in Table 3.2. The three designs have been implemented in the Verilog language and the hardware synthesis to measure the area is performed using the standard design analyzer tools from Synopsys [35]. To make an evenhanded comparison of our design with the architectures already implementing a large register file (with same number of registers as the combination of main register bank and the ARC unit), we measure the overhead of the area only due to additional logic (CSAs, Add/Sub units and routing) required to perform matrix multiplication computation. The corresponding results are shown in the third column of the table. The table shows that the area overhead we incur, due to the added logic for matrix multiplication, is significantly less. This is due to the fact that the LUTs are consuming more than 80% of the ARC unit area, and this agrees with the current processor designs which often devote a largest fraction (up to 80%) of on-chip transistors to caches.

Table 3.2 On-chip area for ARC design using 3-LUTs, implemented at 0.18μ technology. Area overhead is ARC unit area excluding LUTs (registers).

Config. time	*ARC area (square microns)*	*Area overhead (square microns)*
2 cycles	233017	42419
4 cycles	199509	32063
8 cycles	178992	26224

3.3.2 ARC as an Additional Register Bank

A lookup-table (LUT) is a segment of SRAM, e.g. a 3-LUT is an SRAM logic with a bitline width of eight cells. The width of the wordline can be designed according to the functional requirements. A wordline in the LUT is read or written by selecting the wordline using a decoder logic, similar to the implementation of read and write accesses in on-chip memories. When the ARC unit acts a register bank, it can be considered to be consisting of eleven register sub-banks (corresponding to eleven 3-LUTs), each sub-bank comprising of eight registers. Correspondingly, each sub-bank consists of one read and one write ports. The design of one such register sub-bank is shown in Figure 3.6. The flag sets the ARC unit into the register file mode or the computing mode. According to the flag setting, the address and data values appropriate for the current operation are selected. The access time for a register, i.e., to read a value from one memory bank, for each of the designs is shown in Table 3.3. The access time values have been computed at 0.18μ technology, using a register file access time model derived from CACTI [80, 63]. It can be observed that for an implementation with maximum possible configuration cycle time (for a 3-LUT based design it is 8 cycles, for a 4-LUT based design it is 16 cycles, for a 5-LUT based design it is 32 cycles), the register access time is slightly less than a corresponding design with reduced configuration cycle time. This is mainly due to the absence of a decoder and multiplexing logic to select one of the outputs from the segments of an LUT.

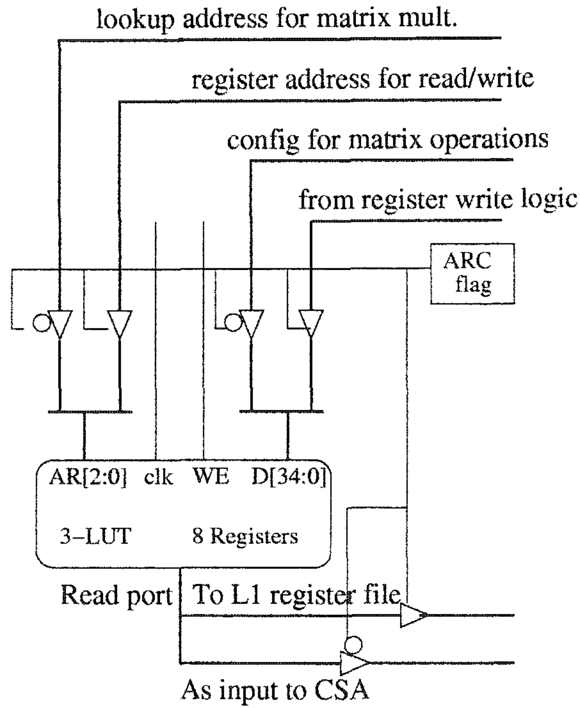


Figure 3.6 A register sub-bank in the ARC unit.

Table 3.3 Register access time in the ARC unit at 0.18μ technology.

ARC config time	*Register access time (ns)*		
	$3-LUT$	$4-LUT$	$5-LUT$
2 *cycles*	0.4484	0.4975	0.5466
4 *cycles*	0.4524	0.5017	0.5510
8 *cycles*	0.3628	0.5098	0.5594
16 *cycles*	-	0.3831	0.5756
32 *cycles*	-	-	0.4028

3.4 Performance Analysis

Table 3.4 Various processor configurations simulated.

Index	*Configuration*	Number of registers available	Computing bandwidth available
C1	Base processor without ARC unit	256 registers (access time 1 cycle)	8 Integer ALU and 4 Integer Multipliers
C2	Base processor with ARC unit as register bank	256+88 registers (access time 2 cycles)	same as base processor
C3	Base processor with ARC unit as computing unit	256 registers (access time 2 cycles)	ARC unit, 8 Integer ALU, and 4 Integer Multipliers

We used Simplescalar-3.0 [9] to simulate a dynamically scheduled out-of-order issue superscalar processor with the simulation parameters summarized in Table 3.1. The configurations for three different processor organizations used for the analysis purposes are as shown in Table 3.4. Configuration C1 is a base processor with out the ARC unit. Configurations C2 and C3 are the base processor with an embedded ARC unit. The performance of the processor with ARC unit acting as an additional register bank is analyzed in the configuration C2. Similarly, performance of the processor with ARC unit acting as a matrix multiplication unit is analyzed in the configuration C3. For the C1 and C2 configurations, matrix multiplication operation is performed in a conventional superscalar processor with varied register file sizes and register access times as shown in Table 3.4.

The register access time for C2 configuration is taken to be one cycle more as compared to the base processor, due to the larger register file. The pipelining of the register file is not considered for analysis purposes here, though the multiple-cycled pipelined register file designs have been proposed in the recent past. To perform the matrix multiplication in the processor with C3 configuration, the entire computation is broken into multiple threads. The benchmark program is developed such that close to two-thirds

of the number of threads are used to generate instructions to be executed in the conventional functional units. The rest of the computation is used to generate instructions to be executed in the ARC unit. During the computation in the ARC unit, the time to configure the LUTs for each element of A is taken to be 4 cycles. A slightly higher speedup in computation can be obtained if a 2 cycle configuration time is used, while a slightly lesser speedup is obtained if a 8 cycle configuration is used. In C3 configuration, even when the register file size is same as that of the base processor, the access of each register involves decoding and thus includes the ARC unit also. Hence, in this case the register access time is taken to be 2 cycles.

For a comparative evaluation between the configurations C2 and C3, where C2 has larger memory bandwidth while C3 has higher computing bandwidth, the multiplication of matrices of size 256x256 is also performed according to the *blocking algorithm* [26, 83]. Blocking is used to achieve locality in the on-chip memory bandwidth available. To perform matrix multiplication by blocking, we divided the 256x256 matrix into four 128x128 blocks.

The speedups obtained by computing the matrix multiplication product in the superscalar processor with an embedded ARC unit as compared to the computation in a base processor are shown in Figures 3.7, 3.8, and 3.9. For each matrix size, the total number of cycles taken to execute the function in the processors with configurations C2 and C3 is normalized with the total cycles for execution in the base processor (configuration C1). In the case of matrix multiplication by blocking, the execution times obtained for configurations C1, C2 and C3 are normalized with the execution time obtained with matrix multiplication in a conventional way in processor configuration C1. As seen in Figure 3.8, for multiplication of matrices with smaller sizes, the C2 and C3 configurations perform similarly and better than the base configuration.

As the matrix size is increased to 128x128, the demand for higher computing power takes precedence and hence the C3 configuration performs better. However, as the matrix size is further increases to 256x256, the demand for larger memory bandwidth is more than the demand for the higher computing power. Thus, in this case though C3

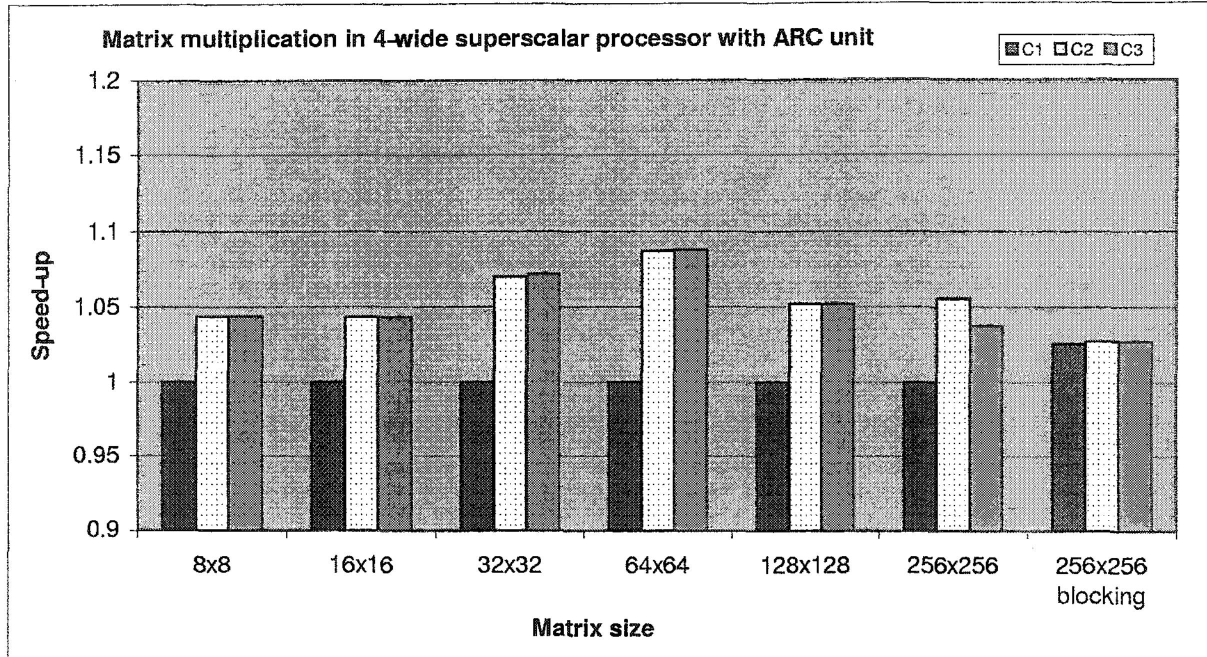


Figure 3.7 Matrix multiplication in 4-wide out-of-order superscalar processor without ARC unit, and with ARC unit

performs better than C1, processor configuration C2 performs even better. In the case of multiplication of matrices by blocking, the configuration C3 performs better than C2 and C2 performs better than C1. This is due to the fact that, blocking helps in utilizing the memory bandwidth efficiently, and thus the demand for computing bandwidth is more than the demand for the memory bandwidth.

From the above results, and especially analyzing the performance of various configurations of a 8-wide superscalar processor in executing multiplication of matrices of sizes 128x128, and 256x256 with and without blocking, it can be concluded that applications depending on their nature, demand either higher computing capacity or larger memory bandwidth or both. Hence, providing memory and computing resources on-chip that are fixed in nature is expensive and does not enable an efficient utilization of silicon real estate on-chip. Instead, it is more beneficial to design a portion of on-chip resources to be reconfigurable, so that it can be used either as a memory element or a computing unit, as the situation demands.

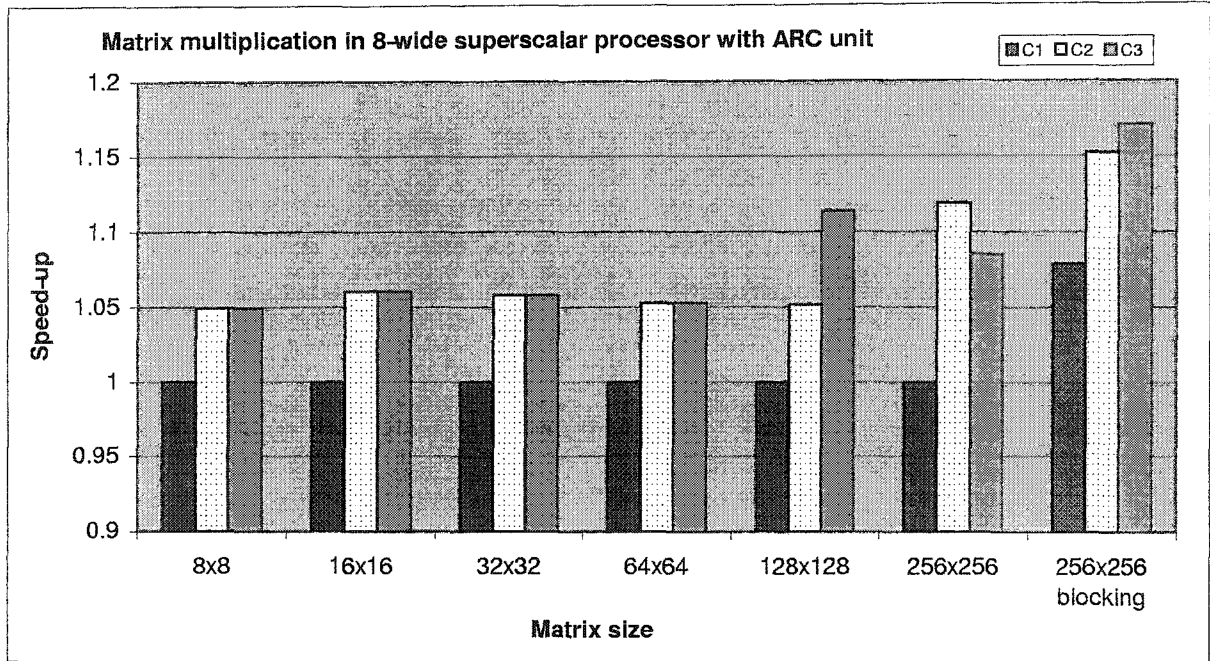


Figure 3.8 Matrix multiplication in 8-wide out-of-order superscalar processor without ARC unit, and with ARC unit

3.5 Summary and Future Work

Applications that demand either higher on-chip computing power or larger on-chip memory bandwidth are continuously emerging. Processor researchers and designers are continuously facing the uphill task of simultaneously meeting these two demands by various applications. In this chapter, we proposed *Adaptive Register file* architecture, a novel architecture to provide a feasible solution and address the above problems concurrently. The *Adaptive Register file Computing* (ARC) unit provides a higher on-chip computing capacity by executing a compute-intensive function, and provides a larger register file resources to meet the memory bandwidth requirements. Results show a performance increase of up to 12%, when an out-of-order 8-wide issue superscalar processor is supplemented with the ARC unit to process matrix multiplication, a compute-intensive core function in most multimedia applications. Similarly, a 17% performance enhancement is seen when the matrix multiplication is performed in an out-of-order 16-wide issue

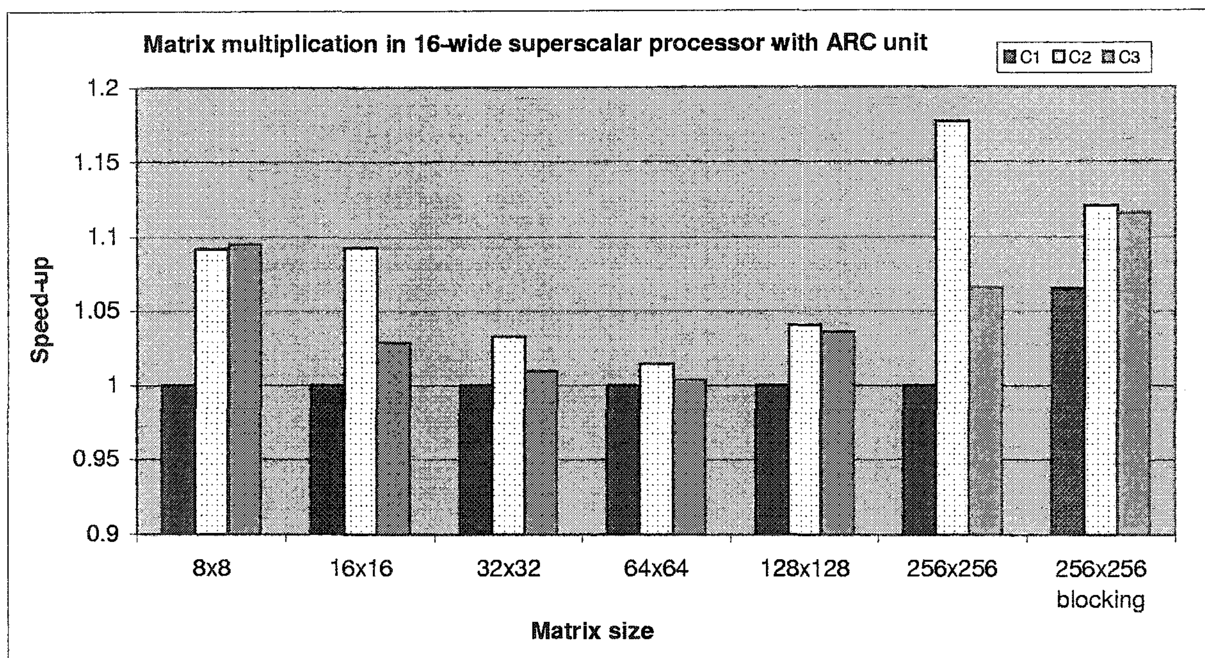


Figure 3.9 Matrix multiplication in 16-wide out-of-order superscalar processor without ARC unit, and with ARC unit

superscalar processor supplemented with the ARC unit. The chapter also discussed the microarchitecture level details for the implementation of the ARC unit.

Further investigations involve the study of performance of proposed architecture in executing various multimedia and signal processing applications that have the matrix multiplication as the core compute-intensive function. The study also requires the development of a compiler that automatically extracts the matrix multiplication function in an application and generates a suitable sequence of instructions to perform the computation in the ARC unit. A mechanism, with a suitable support at the compiler and the microarchitecture level, needs to be devised to support the dynamic allocation of the ARC resources for computation and register file purposes.

4 LOW-POWER HIGH-PERFORMANCE ARCHITECTURES

It is important to ensure that all the hardware resources available on the chip be utilized to the maximum extent possible for wide range of applications. For example, current processor designs often devote a largest fraction (up to 80%) of on-chip transistors to caches. However, many workloads, like the media processor applications, do not fully utilize the large cache resources due to the streaming nature and the lack of temporal locality for the data. On the contrary, these multimedia applications require more computing resources due to the fact that they primarily are compute intensive functions.

From the above observations, an idea of a different kind of computing machine - Reconfigurable Functional Cache (RFC) based Adaptive Balanced Computing (ABC) architecture, has evolved. RFC based architecture uses a dynamic configuration of a part of on-chip cache memory to convert it into a specialized computing unit that can carry out an independent computation. An RFC module operates as a conventional cache memory module or a specialized computing unit [41]. The first version of the RFC based ABC microprocessor [42] has proved the utility of embedding RFCs in the superscalar processor, by accelerating the media applications with speedups in the range of 1.04x to 5.0x. Since cache access is in the critical timing path of the processor, it is important to address the impact of making the cache reconfigurable on the cache access time. In this chapter, we show that even with the area overhead the access time of RFC is same as the conventional cache at larger cache associativities due to the parallel decoding mechanism employed.

Further, due to the increased concern of power dissipation in the system, current mi-

croprocessors are required to deliver higher performance even while keeping the levels of power dissipation at minimum. Earlier, power dissipation was an issue only for designers of embedded and portable systems due to the necessity of prolonging the battery life. Now, it has also become a primary concern for the design of high-end microprocessors due to the cost, heat dissipation and complexity involved in the chip packaging. The area overhead caused due to the added routing structure in the cache, to make it a computing element, will result in higher power dissipation in RFC as compared to a conventional cache. However, the reduced number of cache accesses and lesser utilization of other on-chip resources, due to a significant reduction in the execution time of the application, will result in significant savings in energy consumption. In this chapter, we demonstrate by using the simulation results obtained from executing various media benchmarks on the ABC processor that there is savings in the overall energy consumption.

The rest of the chapter is organized as follows. In Section II, the organization of ABC microprocessor is discussed, with the implementation details of RFC modules at the microarchitecture level. The section also addresses the issues related to the impact on the cache access time and energy dissipation. In Section III, we provide a brief overview of the architecture simulator used to implement the ABC architecture. Section IV presents the performance of the ABC processor as compared to a conventional processor for various media benchmarks. In Section V, we discuss the proof for our claim of significant savings in the power consumption, along with the details of power models used. Finally, Section VI concludes the discussion.

4.1 ABC Microprocessor

In the following sections, we provide the details of the first version of the RFC based ABC architecture to lay the foundation for the further discussion and analysis regarding the power and performance of the processor.

The ABC architecture is built by incorporating a multiple-way set associative data cache memory in a RISC superscalar microprocessor. Some modules in the set associative data cache are built as RFCs. One possible configuration of an ABC microprocessor

with a 4-way RFC is shown in Figure 4.1. Each RFC module can be configured to a specialized computing function or can be used as a normal data cache memory module. The RFC module is constructed as a two dimensional array of multibit output Lookup Tables (LUTs). Each LUT acts as a basic memory element in the memory mode while acting as a basic processing element (PE) in the computing mode.

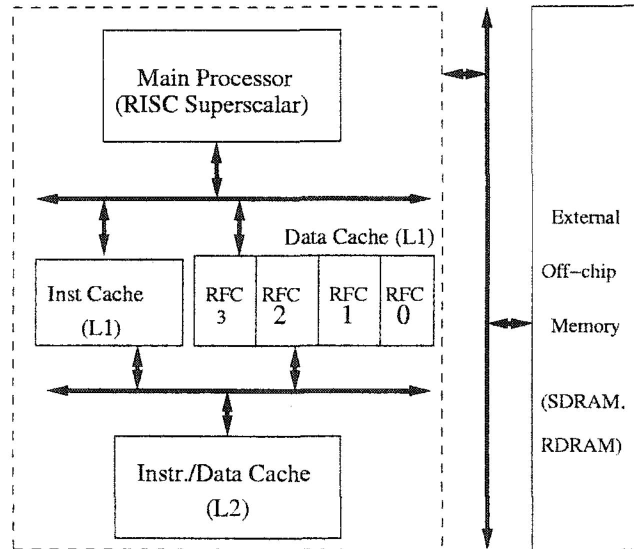


Figure 4.1 ABC: RISC superscalar processor coupled with a 4-way RFC.

4.1.1 RFC Microarchitecture

The components in a conventional cache structure constitute of decoders, data and tag arrays, sense amplifiers (in both the data and tag arrays), comparators, multiplexer drivers, and output drivers. It can be noted that, to convert a conventional cache into an RFC, it is necessary to modify only the organization of data array with addition of the routing structure to facilitate the computation of a function. The organization of the rest of the cache components remains unchanged. Hence, in this section we discuss in detail, only the microarchitecture of the data array in the RFC.

The basic cache parameters are C (cache size in bytes), B (block size in bytes), A

(associativity), and S (number of sets = C/(B*A)). To alleviate the problem of longer-than-necessary access time, Wada et al. proposed the division of the array into subarrays, and presented four parameters, N_{dwl} and N_{dbl} for data array and N_{twl} and N_{tbl} for tag array [75]. The parameter N_{dwl} indicates the number of times the data array is split with vertical cut lines (creating more, but shorter wordlines). N_{dbl} indicates the number of times data array is split with horizontal cut lines (causing shorter bitlines). The total number of subarrays in the data array of the cache is N_{dwl} x N_{dbl}. Similarly, N_{twl} x N_{tbl} is the number of subarrays in the tag array. Along with these parameters, Jouppi et al. used two other organization parameters, N_{spd} and N_{tspd}, which indicate the number of sets mapped to a single wordline in a data array and tag array, respectively [80, 63]. Thus, for the organization of two-dimensional array of LUTs to build the RFC, the parameters N_{dwl} and N_{dbl} are equivalent to the number of rows of LUTs ($= S/N_{lutlines}$) and the number of columns of LUTs ($= 8 * B * A/N_{lutbits}$), respectively. $N_{lutlines}$ is the number of lines in one LUT, 16 for 4-LUT, while $N_{lutbits}$ is the width of one line in an LUT. The structure of the data array in an RFC module is shown in Figure 4.2. The organization of the wordlines, bitlines and the parallel decoding structure in the RFC module had been extensively discussed in [41]. In this chapter, we concentrate more on the impact of such organization on the cache access time and energy dissipation, the two critical parameters in a processor design.

In a 4-way RFC, consider a situation in which one module is active as a computing unit while the other three modules are serving as data cache. During the read and write accesses, the data and tag are read from the sets of all the modules. The bitlines and wordlines are active in the computing module too. However, the set that is selected in the computing module will behave different from the other modules, according to the index decoding performed for set selection. It can be observed from Figure 4.2 that lower four bits of index is replaced with the data word from routing structure, in the module acting as a computing unit. Hence the set selected is different from the sets selected in the other modules. The data read through the wordlines and bitlines in the computing module are not sent to the CPU by blocking at tag comparison level. This can be done by forcing

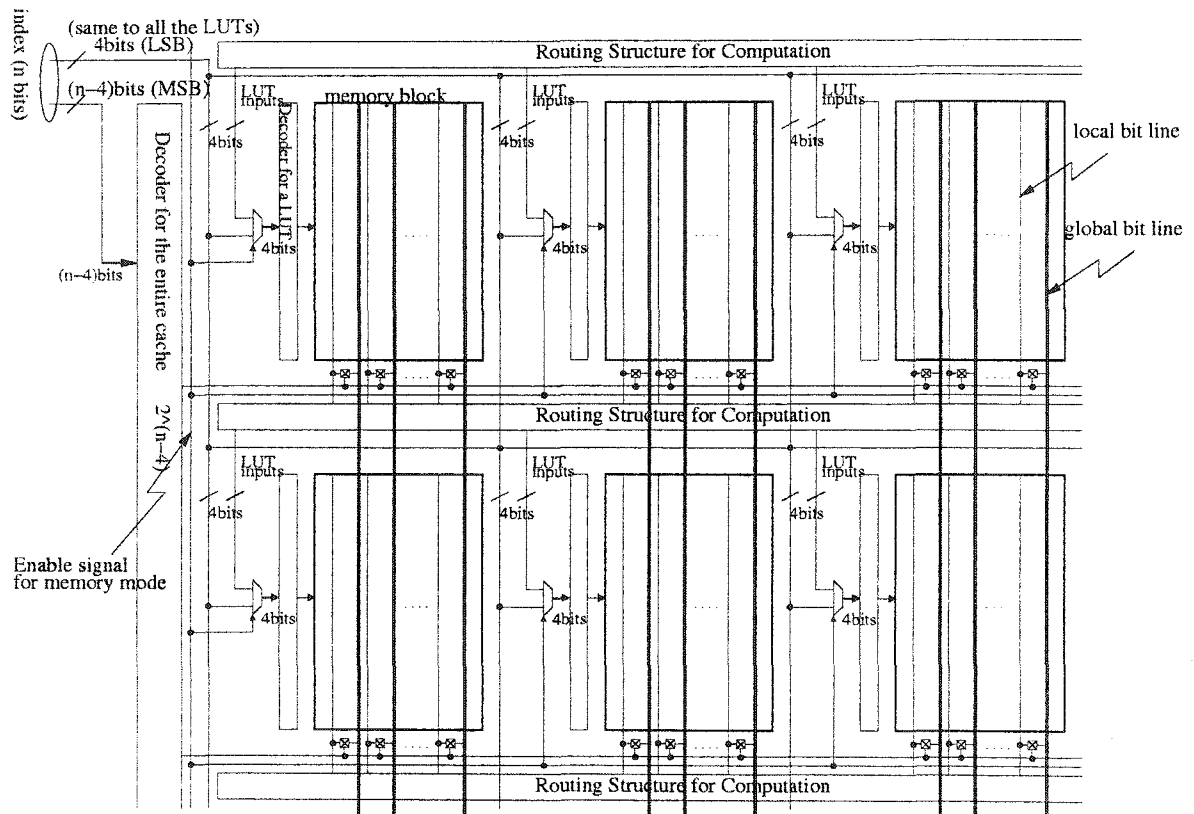


Figure 4.2 Data array structure in one module of A-way set associative RFC. The data array in other modules is similarly arranged

the result of the tag comparison to a miss, with the help of RFC flag, for that particular module. Therefore, even when three modules are active as data cache and the fourth module is active for computation, data accesses are processed by all the four modules. Thus, to estimate the power consumption in RFC due to cache memory accesses, all the four modules are accounted for contributing towards the energy dissipation. The RFC configuration is loaded into the module at the beginning of activation using the conventional load instructions. These load instructions will be a miss in the cache and hence a cache load operation is performed for each of these instructions. Such loads are also processed by all the four modules, in terms of decoding, tag comparison and other necessary operations. However, configuration data is loaded into only one module using the cache replacement strategy. Again, to estimate the power consumption in RFC due to the configuration load instructions, all the four modules are accounted for contributing towards the energy dissipation. A detailed discussion on the configuration and computation modes of the RFC is given in the following sections.

4.1.2 Computing Structure in RFC Modules

The routing structure for computation laid out between two rows of LUTs, as shown in Figure 4.2, facilitates the computation of a function by allowing flow of execution data from one row of processing elements (PEs) to the other. Multiply-and-Accumulator (MAC) and Distributed Arithmetic (DA), the two primitive functions in DSP and the multimedia applications, are implemented in RFC modules. Discrete Cosine Transform (DCT) is the most efficient technique in image encoding and compression schemes. Convolution, a DSP algorithm, is another common requirement in signal and image processing for pattern recognition, edge detection, etc. Using these two algorithms to map to RFC modules, we can implement the most common multimedia applications like mpegdecode, mpegencode, cjpeg from UCLA mediabench [48] and FIR, IIR from TMS320C6000 benchmarks [36].

The number of pipeline stages for the convolution in an RFC depends on the size of

the cache. A conventional convolution algorithm (FIR) is shown in Equation 4.1,

$$y(n) = \sum_{k=0}^{L-1} h(k)x(n-k), \qquad n = 0, 1, 2, ..., \infty \tag{4.1}$$

where, L is the number of taps and h(k) is the constant coefficient. Similarly, the DCT/IDCT function, which is the most effective transform technique for image and video processing is shown in Equations 4.2 and 4.3.

$$Y = C^T AC. \tag{4.2}$$

$$A = CYC^T. \tag{4.3}$$

where, A is a square image (or a square portion of a large image), C is a constant coefficient matrix with the same dimensionality as A, and C^T is a transpose of C. Consequently, Y is the transformation matrix.

The one stage of convolution, consisting of a multiplier and an adder, is shown in Figure 4.3. The first rows of LUTs implement an 8-bit constant coefficient multiplier, as a two 4x8 partial products, and the second row implements the addition of the partial products. The next two rows implement a 24-bit adder to accumulate up to 256 taps of FIR filter. In one module (way) of a cache with 256 sets, we can implement four such stages. Similarly, the implementation of two PEs of DCT/IDCT in four rows of LUTs in an RFC module is shown in Figure 4.4. In both the functions, the carry select adder scheme is used to implement a faster addition operation, thus minimizing the propagation delay for the entire operation. Thus, the routing structure for computation consists of only the registers, bus lines and 2-to-1 multiplexers to enable the data flow between rows of LUTs. To address in detail various issues related to the performance of RFC and its impact on the microprocessor in this chapter, we avoid the discussion on the details of the LUT-based computational implementation of the above functions. An extensive discussion on the implementation of the functions in LUT based RFC is given in [42].

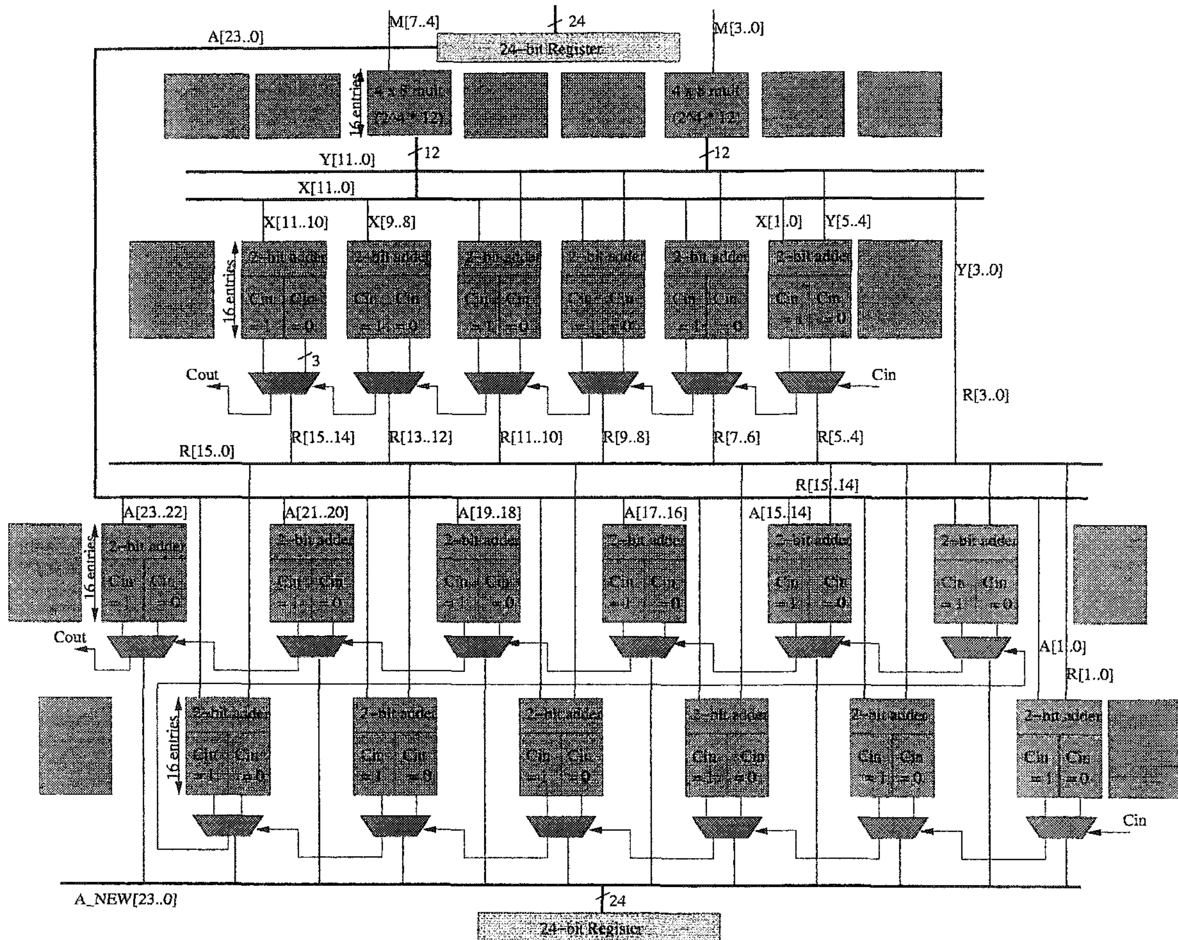


Figure 4.3 One tap of FIR filter implemented in four rows of LUTs in an RFC module with 8 columns of LUTs. The blank LUTs do not participate in computation mode, but function normally in memory mode.

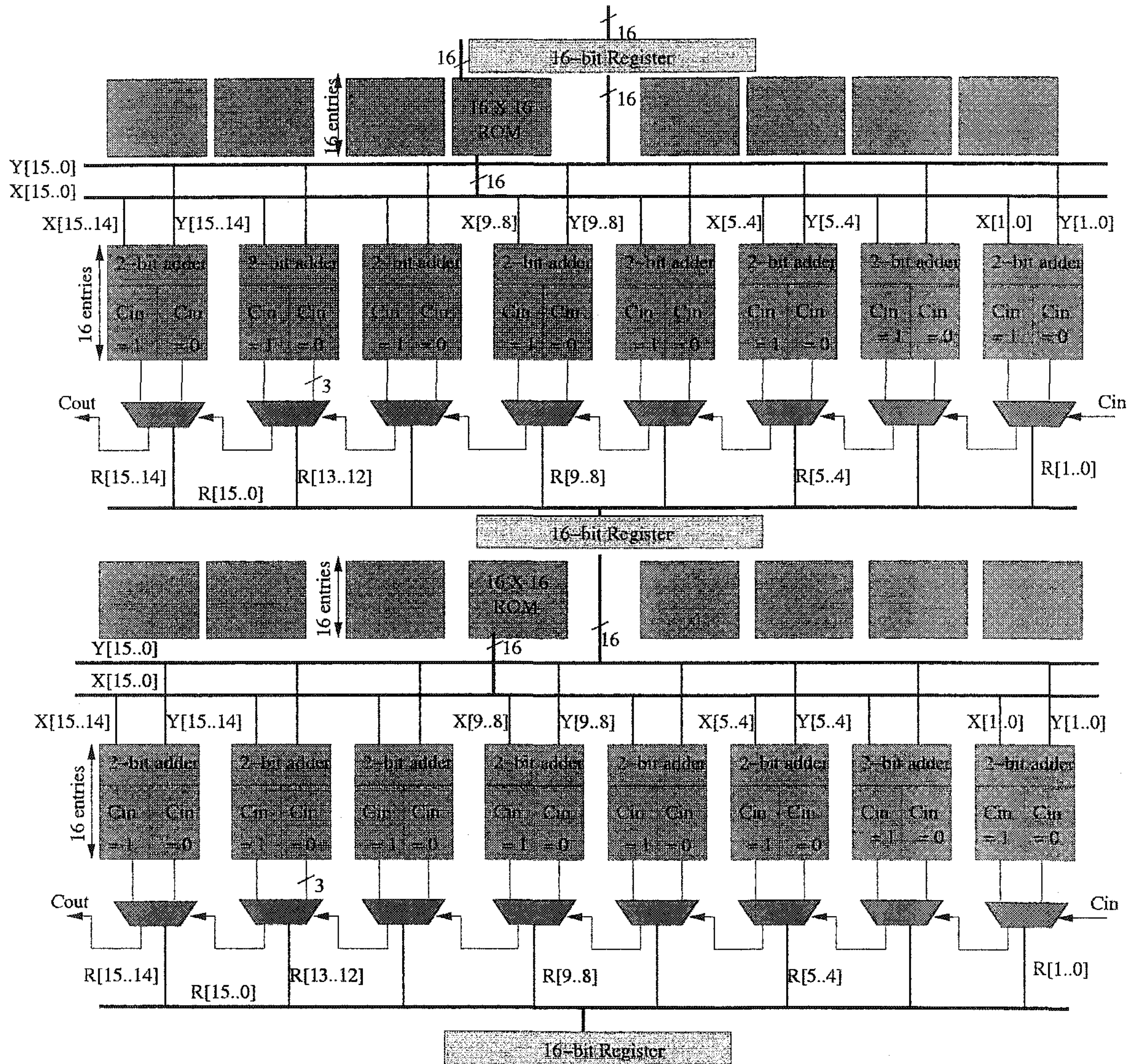


Figure 4.4 Two successive processing elements (PEs) for DCT/IDCT, implemented in four rows of LUTs in an RFC module with 8 columns of LUTs. The blank LUTs do not participate in computation mode, but function normally in memory mode.

4.1.3 Access Time and Energy Dissipation

The propagation delay in various components in a cache contribute towards the cache access time, while the switching activity in them for each cache access contributes towards dynamic power consumption in the cache. These components can be identified mainly as decoder, wordlines (in both data and tag arrays), bitlines (in both data and tag arrays), sense amplifiers (in both data and tag arrays), comparators, multiplexer drivers, and output drivers (data output and valid signal output). The organizational parameters discussed in Section 4.1.1 influence access time and energy dissipation, as variation in each of these parameters results in the requirement of varying number of components. For example, increasing N_{dbl} or N_{spd} increases the number of sense amplifiers required, while increasing N_{dwl} results in the requirement of more wordline drivers. To measure access time and energy dissipation in RFC as compared to a conventional cache, we have used CACTI models [80, 63], with necessary modifications to account for the overhead caused due to the routing structure for computation. The CACTI models estimate cache access time and energy dissipation per access for each set of values for the parameters N_{dbl} and N_{dwl}, while varying other organizational parameters like C, A, and B. However, for the RFC, parameters N_{dbl} $(= S/N_{lutlines})$ and N_{dwl} $(= 8*B*A/N_{lutbits})$ vary according to the variation in each of the organizational parameters C, A, and B. It is to be observed that, as the parameters are varied across a spectrum of cache sizes and associativity, there would be a steady increase in the number of decoders in the data array. We incorporated these necessary features in the model for the estimation of access time and energy dissipation in RFC. The values for data array parameters N_{spd}, N_{dwl} and N_{dbl} in the conventional cache are chosen as 1, 2, and 4, respectively. The values for tag array parameters N_{tspd}, N_{twl}, and N_{tbl} are chosen as 2, 1, and 2, respectively. These are the best values obtained from the simulation results in [63] for optimum access time and energy dissipation in the cache. Figure 4.5 shows the estimation of access time, in nanoseconds, for base cache and RFC, with varying cache organizations. The estimation of energy dissipation per cache access, in nano Joules, is shown in Figure 4.6. In each of these comparisons, an RFC with implementation of the convolution algorithm, is considered.

Similar results are obtained for an RFC with the DCT/IDCT implementation. In a set-associative RFC, when one of the modules is active as a computing unit, the other (A-1) modules will be in memory mode. For this reason, direct-mapped cache is not considered for the implementation of RFC. Hence, the results in Figures 4.5 and 4.6 are shown for 2- to 32-way set-associative caches with varying sizes.

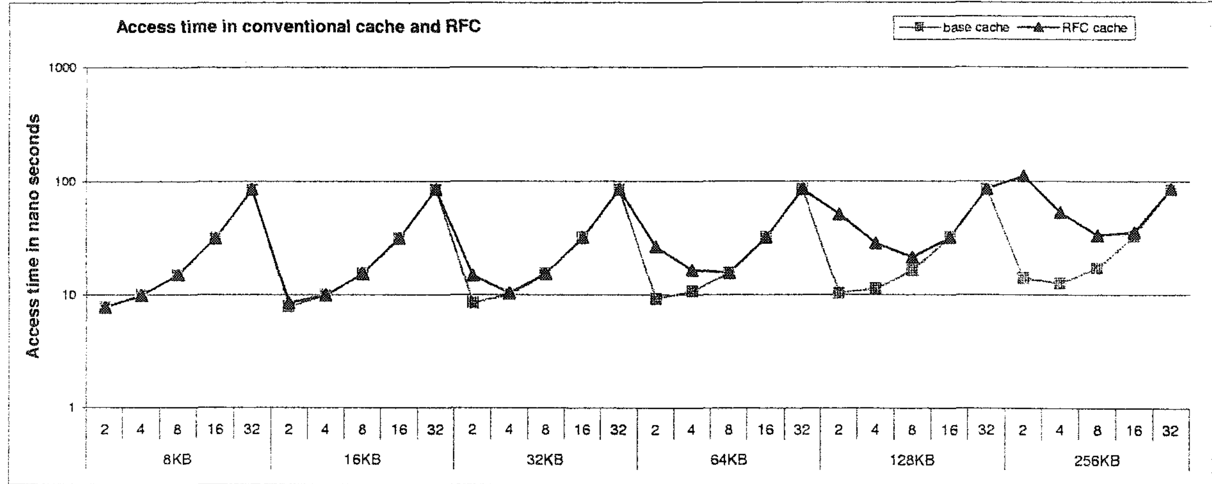


Figure 4.5 Access time, at 0.8 micron CMOS technology, in conventional cache and the RFC for Convolution algorithm implementation. x-axis indicates the cache associativity from 2- to 32-way, for each cache size in KB.

When cache size is increased, access time grows due to the increased delay in decoder, bitline, wordline, and multiplexer driver. Increase in associativity of the cache increases delay in the tag path constituting of tag comparators, tag bitline, tag wordlines etc., and subsequently increases the cache access time. In RFC, routing structure between the rows of LUTs causes an additional delay, apart from the components mentioned above. Since, we have assumed a line size of 32B for all cache sizes, number of sets for caches with lower sizes (8KB, 16KB, 32KB) is smaller and hence number of rows of LUTs is small. Thus the delay caused by routing structure is insignificant. At lower associativities for caches with larger sizes (64KB, 128KB, 256KB), number of rows of LUTs is higher resulting in a larger amount of routing structure in one module of the cache. Delay caused

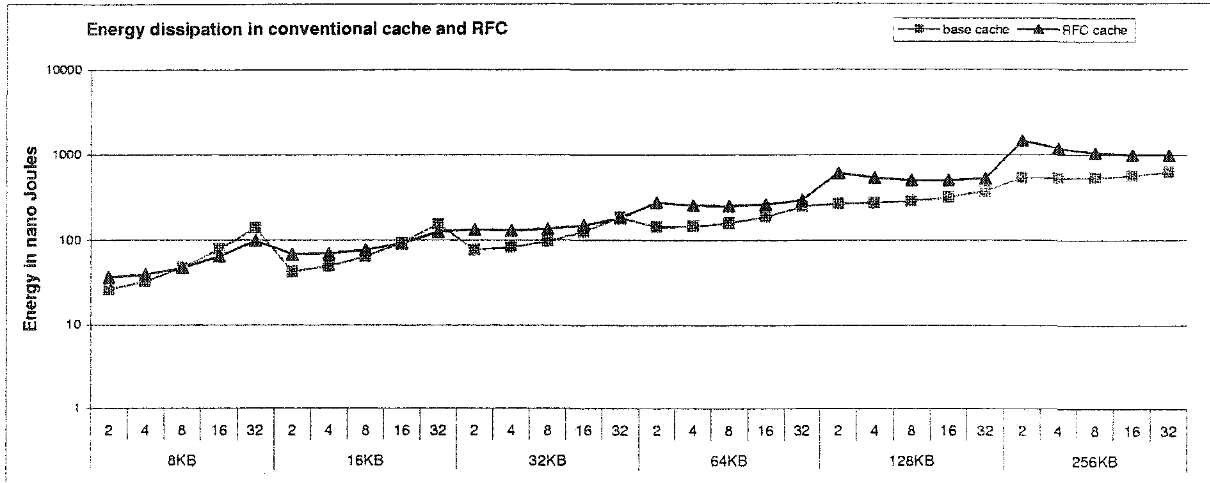


Figure 4.6 Energy dissipation, at 0.8 micron CMOS technology, in conventional cache and the RFC for Convolution algorithm implementation. x-axis indicates the cache associativity from 2- to 32-way, for each cache size in KB.

by the routing structure is larger, and hence larger access time for RFC as compared to conventional cache. As the associativity increases, number of pipeline stages for computation is less and hence smaller delay. Thus at higher associativities, access time in RFC is closer to access time in the conventional cache.

In a conventional cache, a major portion of energy dissipated by the set associative configurations is in bitlines and sense amplifiers. As the associativity increases, requirement of the number of sense amps grows and subsequently power consumption also grows. Besides, increase in the number of subarrays at higher values of N_{dwl} and N_{dbl} results in a larger number of decoders and sense amplifiers. This leads to higher amount of energy dissipation in the RFC. Apart from that, the routing structure contributes towards an additional energy dissipation. At smaller cache sizes, difference in energy dissipation in RFC and a conventional cache is smaller due to the small overhead from routing structure and the number of sub arrays in both the models of cache being close. We assume that the size of an LUT remains constant (in all of our implementations we used a 4-LUT as a basic PE). Therefore, as the cache size increases, number of data

subarrays increases and hence higher power consumption. However in reality, for higher cache sizes, we do not have to convert complete cache into RFC. Only some modules can be used for computational purposes, and the remaining modules can still be designed as conventional cache modules. This will reduce area overhead and consequently overhead in the power dissipation.

From the Figures 4.5 and 4.6, it is clear that the values for the access time and the energy dissipation in an RFC are comparable to those for the conventional cache, at various cache sizes and organizations. The main reason for the deflated overhead for both these parameters in the RFC is due to the fact that only the organization of the data array is modified to build an RFC, while organization is unchanged for the remaining components like tag array, comparators, multiplexer drivers and the output drivers.

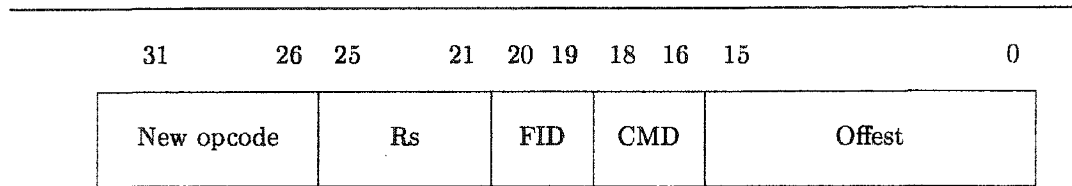
4.2 Architecture Simulator for ABC Implementation

The mechanism of computation in RFC and the functioning of the ABC microprocessor are discussed in detail in [42]. Here, we present a brief overview of the operation of the ABC architecture. For the simulation purposes, the ABC processor is implemented by appropriate modifications of Simplescalar tool set [9]. The source code of the simplescalar tool set is modified to incorporate the RFC design and the microarchitecture is modified accordingly to enable the communication between the main processor and the RFC.

4.2.1 Instructions to Utilize RFC

To prevent the memory address space from becoming non-cacheable, not more than one cache modules is configured as computing units at a time. There are three steps involved to perform a computation on RFC – configuration, loading input, and storing output. All these steps require only load/store class of instructions. To perform these operations, new instructions, named rfc load/store instructions, are added. The format of the rfc instructions is the same as the conventional load/store instructions except for

the target register field. The target register field is used for different purposes as shown in Figure 4.7. Three types of instructions, *rfc_load_conf*, *rfc_load_in*, *rfc_store_out*,



FID (function identifier) : 4 different functions to be implemented into RFCs
CMD (command) : type of operations

000 - start configuration and set the special state register and other required flags

001 - load configuration data (from reserved address space)

010 - end of loading configuration data and set to execution mode

011 - terminate an RFC use for computation and return back to the cache mode

100 - set flags and load input data to be processed for the computation

101 - load 2nd set of input data from memory hierarchy if applicable

110 - store output data to memory hierarchy

111 - set for partial reconfiguration process at the end of current step

RFC instructions for load-word based on the CMD

000 :	rfc_lw_conf_start	*F_class*	special_flags
001 :	rfc_lw_conf	*fid*	offset($r)
010 :	rfc_lw_conf_end	*fid*	offset($r)
011 :	rfc_terminate	*fid*	
100 :	rfc_lw_in1	*fid*	offset($r)
101 :	rfc_lw_in2	*fid*	offset($r)
110 :	rfc_sw_out	*fid*	offset($r)
111 :	rfc_partial_set	*fid*	offset($r)

Figure 4.7 rfc instructions for loading and storing "word" type of data

and initialize/terminate instructions are added. The detailed format of the new class is described in Figure 4.7. It is assumed that at most four RFC modules are available in the architecture. Individual instruction operations are explained briefly later in Section 5.1.2. Note that the FID field chooses the module that will be configured for a specific function. Word-data type *rfc* instructions are used for the description purpose.

For different types of data (for example, half-word or byte, etc.), the same instruction format can be used. Also, note that *rfc_lw_conf_end* and *rfc_partial_set* performs a loading operation in addition to setting the mode as well. The *rfc_lw_conf_start* instruction is used for a special setting in a RFC, such as use of multiple input buffers (i.e. input and intermediate data) and a hierarchical function identification of RFCs, if more than four RFC modules are implemented (e.g. function class followed by sub-FID).

4.2.1.1 Mechanism for the Computation in RFC

In out-of-order issue in a superscalar processor, any instruction which does not have a dependency with preceding instructions can be issued and executed at any time if the required resources are available. In addition, in a speculative execution, the next instruction stream in a code sequence can be executed speculatively. The out-of-order issue and execution may also happen among *rfc* instructions because there is no explicit dependency between *rfc* instructions. However, the *rfc_lw_in*(1 *or* 2) and *rfc_sw_out* instructions must not be issued and executed until the RFC has been configured. A speculative execution mechanism may issue the *rfc* instructions in any order. To avoid this type of exception, a special RFC state register is included. In the register, two bits are reserved for each RFC module. The two-bit RFC state information is organized as follows.

- 00 : NON-RFC/END-RFC - normal cache mode;
- 01 : CONF - configuration mode for RFC
- 10 : CONF_DONE/START-RFC_EXE - end of CONF
- 11 : RFC_EXE - execution mode

The state transition is controlled by the *rfc* instructions as depicted in Figure 4.8. All the *rfc* instructions must access the RFC state register according to the FID field in the microcode and then check the current state with its CMD field. If it is an allowed state, the *rfc* instructions can be issued. Otherwise, the *rfc* instructions are stalled until the corresponding state is resolved.

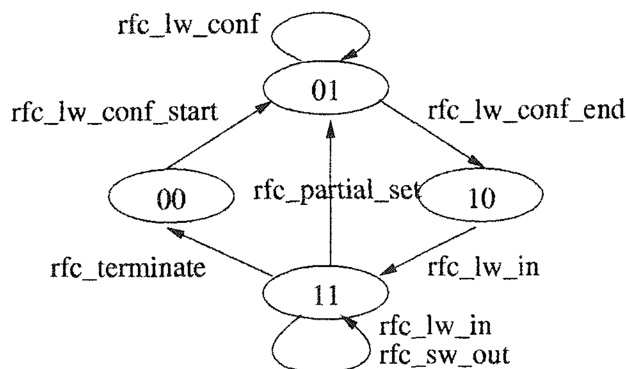


Figure 4.8 State transition for the RFC cache.

4.2.1.2 Configuration

The configuration of RFC from a cache module simply implies loading all the contents of LUTs required to construct a computing unit. A normal cache read with a small modification directs specific data into the designated cache line. The required configuration data for RFCs resides in a reserved memory (address) space in main memory. The configuration is loaded into main memory when the system boots up. The *rfc_lw_conf_start* instruction sets the corresponding RFC state register. The subsequent *rfc_lw_conf* instructions load the configuration lines to the specific RFC without changing the RFC state.

Since the configuration data is held in main memory, the data accesses would be cache misses if the same configuration had not been loaded previously. This cache miss will replace the current clean/dirty lines in a write-back cache. Thus, separate cache flushing is not required. A simple modification of cache replacement mechanism, such as LRU (Least Recently Used), is required to replace the data in the specific cache module (RFC) with the loaded configuration data in a set-associative cache organization. The modified LRU scheme, which is set by *rfc_lw_conf_start*, replaces lines in the RFC only if *rfc_lw_conf* accesses in *CONF* mode. The configuration data in RFC should not be modified by other load/store instructions except the *rfc_lw_conf* instructions if the

RFC state is not '00'. An additional operation in the modified LRU mechanism protects the configuration data by removing the lines in RFC from the replacement line list in LRU when conventional load/store instructions access the cache. The new LRU scheme consists of two operations, one for *rfc_lw_conf* to replace data in the specific RFC and the other for other load/store instructions (including *rfc_lw_in*1/2) to access the rest of cache memory. Using this modification, the set associativity of cache memory is changed dynamically depending upon the use of RFC. A write operation to RFC is prevented by disabling the write enable in the RFC during the execution mode.

4.2.1.3 Execution Stage

The new instructions are decoded in dispatch stage according to their specification described in Section 5.1.1. The *rfc_lw_in*1/2 instruction, which loads input data to an RFC computing unit, is decoupled from LSQ (Load Store Queue) and queued into an input buffer (IBUF) dedicated to each RFC. The decoupled IBUF provides data in-order to the RFC computing unit. In addition, the *rfc* load/store instructions process the input/output data independently in the separate buffers. Otherwise, these instructions pass/receive data to/from RFC serially in one buffer. The effective address for the *rfc_lw_in*1/2 instruction is calculated using the existing datapath and passed to the corresponding instruction in IBUF, not in LSQ. The details of mechanism to support out-of-order update (load) of input data in IBUF are discussed extensively in [42]. If no slot in IBUF is available, the following instructions including conventional instructions fetched from memory are stalled until IBUF is again available. Otherwise, it may require a complex mechanism for a decoupled fetch queue between *rfc* load instructions and other instructions. By queuing the *rfc_lw_in*1/2 instructions into IBUF in-order, the input data to be processed is provided to RFC in a correct order. This is like a reorder buffer mechanism for input data of the RFC unit to remove the impact of out-of-order execution. The input data from memory to the IBUF can be supplied out-of-order as in the conventional LSQ. Note that instead of providing IBUF with the data in write-back stage, the data may be directly loaded into the matched address slot in IBUF from data

buses.

A whole function in an application may not be mapped to an RFC as a computing unit at one time. For instance, in an FIR filter, if the number of taps for the filtering coefficients is larger than the number of physical taps implemented in an RFC, we configure the first set of taps in the RFC and then reconfigure it partially for the next set of coefficients at run-time. This can be achieved using a cache write operation in *rfc_lw_conf*. The *rfc_partial_set* instruction shown in Section 5.1.1 sets the RFC state register and all the required flags as *rfc_lw_conf_start* does. This mechanism protects the current configuration data to be retained in the RFC for the partial reconfiguration by setting the RFC state from *RFC_EXE* to *CONF* mode directly. The following *rfc_lw_conf* instructions reconfigure the designated RFC as done for the initial configuration. In this case, an input data stream may be processed by a fixed number of taps and partial results are stored. In the second iteration, input and partial results are loaded (using *rfc_lw_in*1 and *rfc_lw_in*2, respectively) and processed together. Thus, we provide two IBUFs. The computation in RFC is processed when both data elements are available.

After a computation is completed in the RFC, the output data is queued into an output buffer (OBUF). The OBUF is a simple FIFO register file since the queued data is already in-order. The output data in OBUF is stored into memory by the *rfc* store instructions. The *rfc_terminate* instruction sets the RFC state into the non-RFC mode after finishing an entire computation. This setting should be done in commit stage to avoid mis-execution of pending *rfc* instructions. If the same computation with the current configuration is performed in the near future, the RFC state (*RFC_EXE*) is not changed. The *rfc_lw_in*1/2 instructions do not affect any state when a misprediction/speculation or an exception occurs because the *rfc_lw_in*1/2 instructions do not modify the precise state in microarchitecture. All the repair to be done in this case is to flush the instructions in IBUF as done in a conventional LSQ.

4.3 Performance of ABC Processor

The performance of the ABC architecture is assessed based on the number of cycles it takes to execute each application as compared to a superscalar processor with a conventional cache. The performance of the ABC processor has been discussed extensively in [42] with varying cache parameters like cache size, line size and associativity. A set of those results, for a 32KB 4-way with a line size of 32B, for various media applications are shown in Table 4.1. It is to be observed that, when one of the RFC modules is active as a computing unit, the L1 data cache size available for the application execution is reduced to (A-1) modules. Hence, there is scope for an increase in the cache miss rate. For this reason, [42] gives an elaborate discussion on the increment in the cache miss rate and increased accesses to L2 unified cache, and the subsequent impact on the performance of the processor.

Table 4.1 Comparative performance results of ABC processor and base processor for various benchmarks, for a 32KB 4-way cache with a line size of 32B

	MPEG2 decoding		MPEG2 encoding		CJPEG	
Parameter	Base proc w/o RFC	ABC with RFC	Base proc w/o RFC	ABC with RFC	Base proc w/o RFC	ABC with RFC
instructions executed	3379270939	765200387	1836846152	1557343290	388146976	349668078
instructions committed	2917901442	697388230	1133806586	990419802	313470898	274783704
branches committed	358154116	117537382	193040786	179287391	45744790	46040223
L1 I-cache accesses	3516411442	789989376	1987089036	1730431526	400090831	361615414
L1 D-cache accesses	556436572	243376327	396899127	375379770	88372011	88376692
L2 cache accesses	3105242	2110077	6368541	7464894	1097261	1237426
load instructions executed	529245151	170407053	526670201	483560722	83909135	83848718
store instructions executed	125747787	108910724	26082798	19361466	26250519	26556238
total cycles for application	1112897758	282349760	748791538	659330287	138079889	130595007
total cycles for core function (DCT/IDCT)	860150575	31364247	51600132	2096111	11561504	4410289
load insts. To configure RFC	0	256	0	2162689	0	256
load insts. for data exec inRFC	0	16588800	0	1081344	0	2359296
	FIR 16 taps		FIR 256 taps		IIR	
Parameter	Base proc w/o RFC	ABC with RFC	Base proc w/o RFC	ABC with RFC	Base proc w/o RFC	ABC with RFC
instructions executed	22003487	14051176	139968277	17876122	18090987	14743584
instructions committed	18745647	11217583	136710448	15041687	14734154	11911083
branches committed	3212525	2672596	11076840	2922558	2819926	2706289
L1 I-cache accesses	21732330	13715206	139696753	17540633	17819750	14406960
L1 D-cache accesses	5633334	3666332	37090638	6618032	4538873	3685913
L2 cache accesses	5932	29054	5965	170652	7762	20957
load instructions executed	6026190	3094478	49279950	5063079	4454012	3192988
store instructions executed	1940578	1416164	9804898	2399924	1531309	1416623
total cycles for application	7007648	4962539	38465368	7924806	6087211	5136357
total cycles for core function (Convolution)	2408919	360901	33866654	3323161	1453998	501776
load insts. To configure RFC	0	352	0	2272	0	512
load insts. for data exec inRFC	0	49152	0	1048576	0	32768

In Figure 4.9, each simulation result data set, for a particular cache structure, consists of four values: (a) the total application execution time in the general purpose superscalar processor without the RFC, (b) portion of time taken for computing the core function in the GPP, (c) total application execution time in the ABC processor integrated with the RFC, and (d) the portion of time taken to execute core function in the RFC. Each of these values are normalized to the total number of cycles for executing the application in the conventional superscalar processor. The performance improvement is obtained from the normalized total cycles for the execution of the overall function and also the core function. For example, the execution time for the MPEG2 decode application is only 25.37% of the time if the application is executed on a conventional superscalar processor. This large speed-up is due to the significant acceleration of the DCT/IDCT function in the RFC. The specialized computing units configured in the RFCs improve the performance of each core function significantly. The most important factors for speed-up are the reduced number of instructions and the acceleration of computation with a specialized unit. Hence, it is observed that the overall speed-up is largely proportional to the fraction of the core function, in the entire application, that is mapped to the RFC.

The MPEG2 encode application has a smaller fraction (6.89% as shown in Figure 4.9) of the core function to be mapped to RFC. Besides, we observe that the number of calls for the core function in the application is far less as compared to that in the case of MPEG2 decode application. Therefore, for execution of MPEG2 encode application in the ABC processor, we use the dynamic configuration scheme, where the RFC module is configured only when call for the core function occurs and the module is moved back to the memory mode when the computational requirement is fulfilled. Thus the RFC module has to be continually configured in MPEG2 encode application, while the module is configured just once in the case of MPEG2 decoding at the starting of the execution of the application. Hence, the number of load instructions to configure the RFC module is higher for the execution of MPEG2 encode application. However, the configuration overhead in this case is much smaller than the product of time for loading one configuration and the number of calls for the configuration. This is due to the fact

that the configuration loading is just a single cache write operation for each line. Thus only the first time the configuration needs to be loaded completely. For the subsequent configurations, only those cache blocks that are corrupted are written, thus minimizing the configuration overhead.

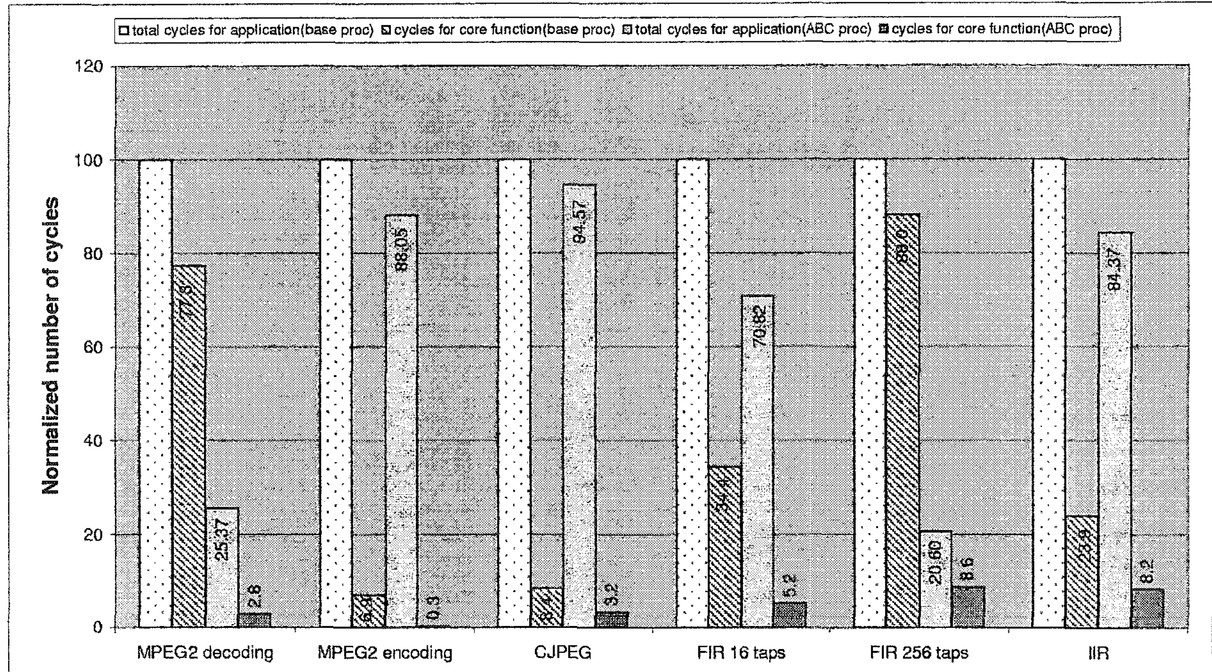


Figure 4.9 Performance of ABC processor vs. base processor for a 32KB 4-way cache with a line size of 32B

4.4 Power Estimation in ABC Processor

In superscalar microprocessors, higher performance is achieved at the cost of higher power consumption since:

- The instruction issue width is increased for higher performance, but employs complex logic. Further, that leads to increased hardware complexity for resolving the instruction dependencies.
- The energy and complexity of the functional units is independent of the issue width. However, with a speculative execution, the functional unit energy per committed

instruction still grows with the issue width because of the increasing portion of instructions that are fetched from mispredicted paths and discarded later.

Higher energy dissipation requires more expensive packaging and cooling technology, increases cost, and decreases reliability of products in all segments of computing market from portable systems to high-end servers. Moreover, higher energy dissipation significantly reduces battery life and diminishes the utility of portable systems. Conserved power consumption is important for various reasons.

- In order to keep transistors within their operating temperature range, heat generated has to be dissipated from the source in a cost-effective manner. Power density may soon limit performance growth due to thermal dissipation constraints.

- Power must be delivered to a very large scale integration (VLSI) component at a prescribed voltage and with sufficient amperage for the component to run. Very precise voltage regulator/transformer controls current supplies that can vary within nanoseconds. As the current increases, the cost and complexity of the voltage regulators/transformers increase as well.

- Batteries are designed to support a certain watt-hours. The higher the power, the shorter the time that a battery can operate.

Until recently, power efficiency was a concern only in battery powered systems like notebooks and cell phones. Currently, increased microprocessor complexity and frequency have caused power consumption to grow to the level where power has become a first-order issue. Each market segment has its own power requirements and limits, making power limitation a factor in any new microarchitecture.

The two sources of power consumption in microarchitectures are:

- Dynamic power consumption:

    * It is the power dissipation whenever a transistor or wire changes voltage (i.e., value).

  * Dynamic power dissipation is proportional to the product of the number of devices changing value, the speed of these changes (i.e., operating frequency) and the square of the voltage change.
  * Reducing power dissipation is possible by reducing each of these factors.

- Leakage power dissipation:
  * Power is dissipated even when devices do not change values due to the imperfect nature of semiconductor based transistors. This is the leakage power.
  * In existing designs, leakage power is relatively small. However, as we move toward smaller transistors and lower voltages, leakage power increases rapidly.
  * Power-aware efforts in this area aim at cutting off power to devices while they are not being used. This is a challenging task as powering on and off devices requires some time and, hence, can severely impact performance.
  * For example, it is possible to reduce leakage power in caches by deactivating parts of the cache with a negligible impact on hit rate and performance.

The earlier sections discussed the high performance of the ABC processor for various multimedia applications. From Table 4.1, it can be observed that, when the core function is executed in the RFC, the total number of instructions executed, the number of load and store instructions and the number of accesses to L1 data and instruction caches have reduced. In this section, we show that the effect of power dissipation overhead per access in the RFC, is offset by the reduced number of accesses to the cache. We further show that the reduced utilization of other on-chip resources, due to a significant reduction in the execution time of the application, will result in savings in energy consumption.

4.4.1 Power Estimation Models

In CMOS microprocessors, dynamic power consumption is the main source of power consumption and clearly dominates the leakage power dissipation. Therefore, we take the

approach of estimating the dynamic power consumption in the ABC processor as compared to a conventional microprocessor based on the simulation results. Architectural power simulators like Wattch [7] measure the utilization of various processor components and during the simulation, feed these utilization numbers into a high-level power model to estimate the energy behavior of the processor. Using a similar approach, we estimate the component power breakdowns based on the utilization numbers we gather from the simulation of various multimedia benchmarks, for both superscalar processor with a conventional cache and the ABC processor. For this purpose, we use two models as discussed in the following sections. The heuristics developed for the estimation of power dissipation in each component are in line with the methodology proposed by Martonosi et al. [7, 37].

4.4.1.1 Alpha Processor Model

The reported component breakdown of power consumption in the Alpha 21264 processor is shown in Table 4.2. The second column (P_{frac}) indicates the power consumed by the component as a fraction of the total power in the processor. The third column indicates the heuristic for the utilization factor of the component.

Table 4.2 Listing of heuristic power estimations for Alpha processor model

Resource	Pfrac	Utilization factor
clock	26.31	total cycles for application
resultbus	2.63	instructions executed
ALU	15.79	instructions executed
L2 cache	2.66	L2 cache accesses
L1 D-cache	12.79	L1 D-cache accesses
L1 I-cache	17.68	L1 I-cache accesses
regfile	2.63	inst. committed –branches comm. + instruction decoded
lsq	6.63	load instrs. + store instrs.
window	8.26	instructions executed
bpred	2.31	branches committed
rename	2.31	instructions decoded

4.4.1.2 Pentium-Pro Processor Model

The reported component breakdown of power consumption in the Pentium-Pro processor is shown in Table 4.3. The second column (P_{frac}) indicates the power consumed

by the component as a fraction of the total power in the processor. The third column indicates the heuristic for the utilization factor of the component.

Table 4.3 Listing of heuristic power estimations for Pentium-Pro processor model

Resource	Pfrac	Utilization factor
Clock	10.5	Total cycles for application
Instruction fetch	18.7	L1 I-cache access
Register Alias Table	4.9	instructions decoded
Reservation Station	8.9	instructions executed
Reorder buffer	11.9	instruction committed
ALU	22.6	instructions executed
L1 Data Cache	11.5	L1 D-cache accesses
L2 Cache	2.5	L2 cache accesses
Memory Order Buffer	4.7	load instructions + store instructions
Branch Target Buffer	3.8	branches committed

4.4.2 Power Estimation in RFC

With the heuristics for utilization factors available, as discussed earlier, the component breakdown of total power dissipation for all the components except the L1 data cache can be obtained. The estimation of power dissipation in the RFC is more involved since while one module of the RFC is acting as a computing unit, the remaining (A-1) modules are serving the memory accesses.

For the estimation of power consumption in RFC under the above mentioned situation, the observations are:

- In a 4-way RFC, even when three modules are active as data cache and the fourth module is active for computation, data cache accesses are processed by all the four modules. Hence, power is dissipated by all the modules for each data cache access. During the data cache access, data from the computing module is avoided from being sent to CPU by blocking the tag comparison. This can be done by forcing the result of tag comparison to a miss, with the help of RFC flag, for that particular module. Thus, to estimate the power consumption in RFC due to cache memory accesses, all A modules are accounted for contributing towards the energy dissipation.

- The configuration is loaded into RFC module, treating the instructions to load configuration as normal load instructions that will undergo a miss in the cache. Hence, a cache write operation is performed for each of these instructions. Thus, cache writes are processed by all the four modules, in terms of decoding, tag comparison and other necessary operations. The stream of configuration data is loaded into one module only using the cache replacement strategy. Therefore, again to estimate the power consumption in RFC due to the configuration load instructions, all A modules are accounted for contributing towards energy dissipation.

- The computation in RFC module is LUT based and lookup cost is proportional to the number of data items processed. Power consumption in RFC module that is active in computing is thus proportional to the number of data items loaded into the input buffer. Also, the data is being processed in all the taps of the computing unit simultaneously. Thus, at any instant of time, data being processed simultaneously in all the taps is equivalent to one memory access.

- From the above observations, it can be seen that computation in one module of RFC is performed simultaneously when the RFC is serving the cache memory accesses. However, for a simple and worst-case estimation of the power dissipation in RFC, we assume that the cache accesses and computation in one module of RFC are separated in time.

The heuristic for power estimation in the RFC of the ABC processor is as given below:

$$Power_{RFC} = Power_{basecache} * Power_{factor} * \left(\frac{M2 + conf_{load} + ((1/A)*(1/N_{taps}) * data_{RFC}}{M1}\right)$$

where,

$Power_{factor}$ = ratio of per access power consumption in RFC to that in conventional cache. This is obtained as in Figure 4.6.

M2 = L1 Data cache accesses in ABC processor with RFC

$conf_{load}$ = Number of load instructions to configure one RFC module

N_{taps} = Number of computing stages in one RFC module

$data_{RFC}$ = number of data elements executed in RFC module

M1 = L1 Data cache accesses in base processor without RFC

A = Associativity of the cache.

4.4.3 Results and Analysis

The component power break down of the power consumption in the ABC processor for both the models for various multimedia applications is shown in Figures 4.10, 4.11, 4.12, 4.13, and 4.14. The power dissipation is estimated for the simulation data shown in Table 4.1, which is obtained for a 32KB 4-way set associative cache with a line size of 32 bytes. It is observed that when the computation intensive core function is mapped and computed in RFC number of instructions executed is reduced. This results in reduced utilization of on-chip resources causing the reduction in overall power consumption. This, along with the reduced number of accesses to data cache, offsets the power dissipation overhead in the RFC. Similar component power break down are obtained for other media applications. Figure 4.15 shows the total power consumed for each of the application under Alpha and Pentium-Pro processor models. The values shown in the figures are normalized to the total power consumed in a conventional superscalar processor for each of the application. The results show that up to 60% reduction in power consumption is achieved for MPEG decoding application, and a reduction in the range of 10% to 20% for various other multimedia applications. The significant savings in the power for the execution of MPEG2 decode application in the ABC processor is due to relatively larger fraction of the application that is mapped to the RFC.

It can be observed that overall power consumption in the processor with RFC is either smaller or almost same as compared to that in the base processor. This is achieved

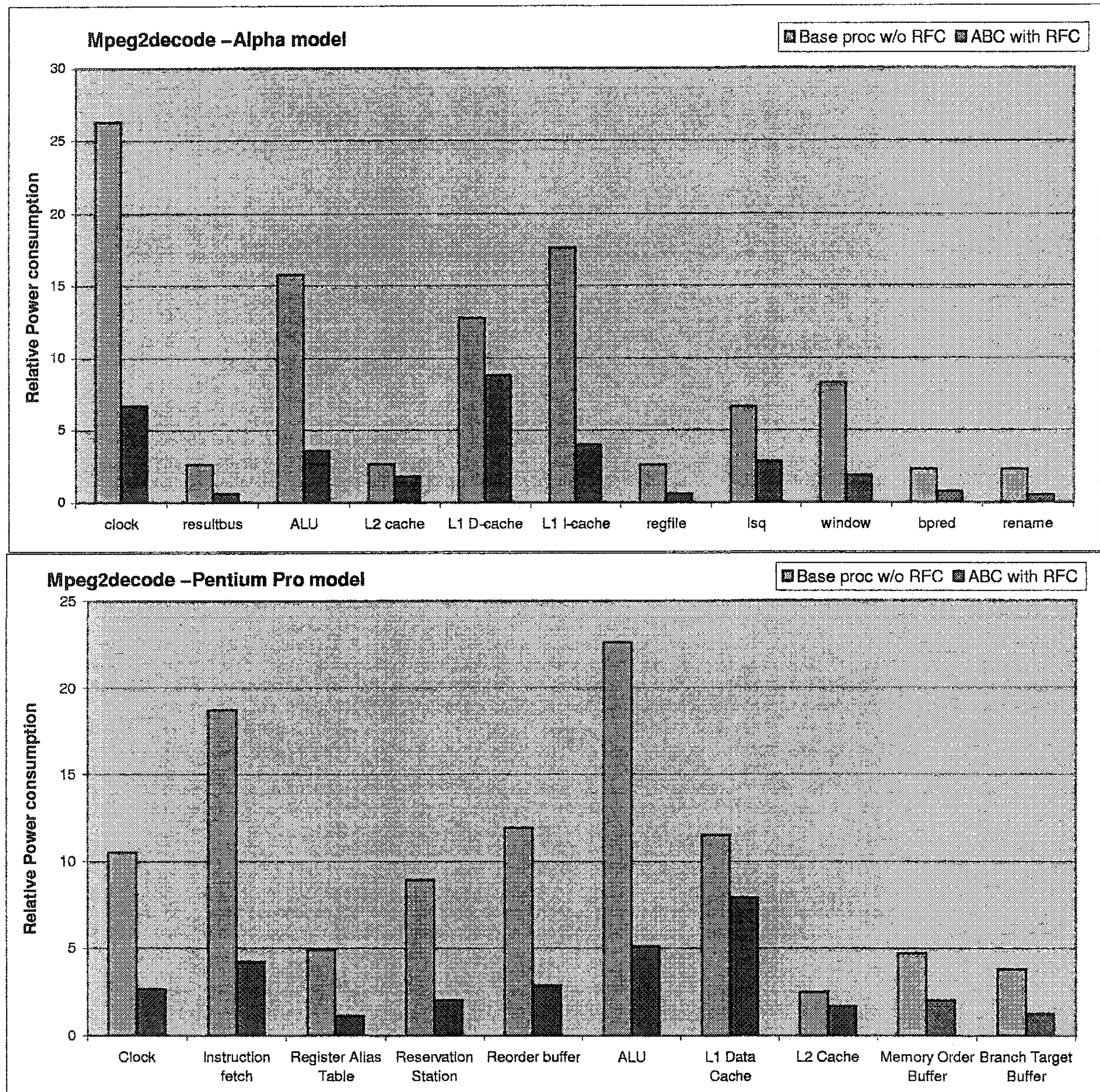


Figure 4.10 Component power utilization in ABC processor vs. base processor for MPEG2 decode application.

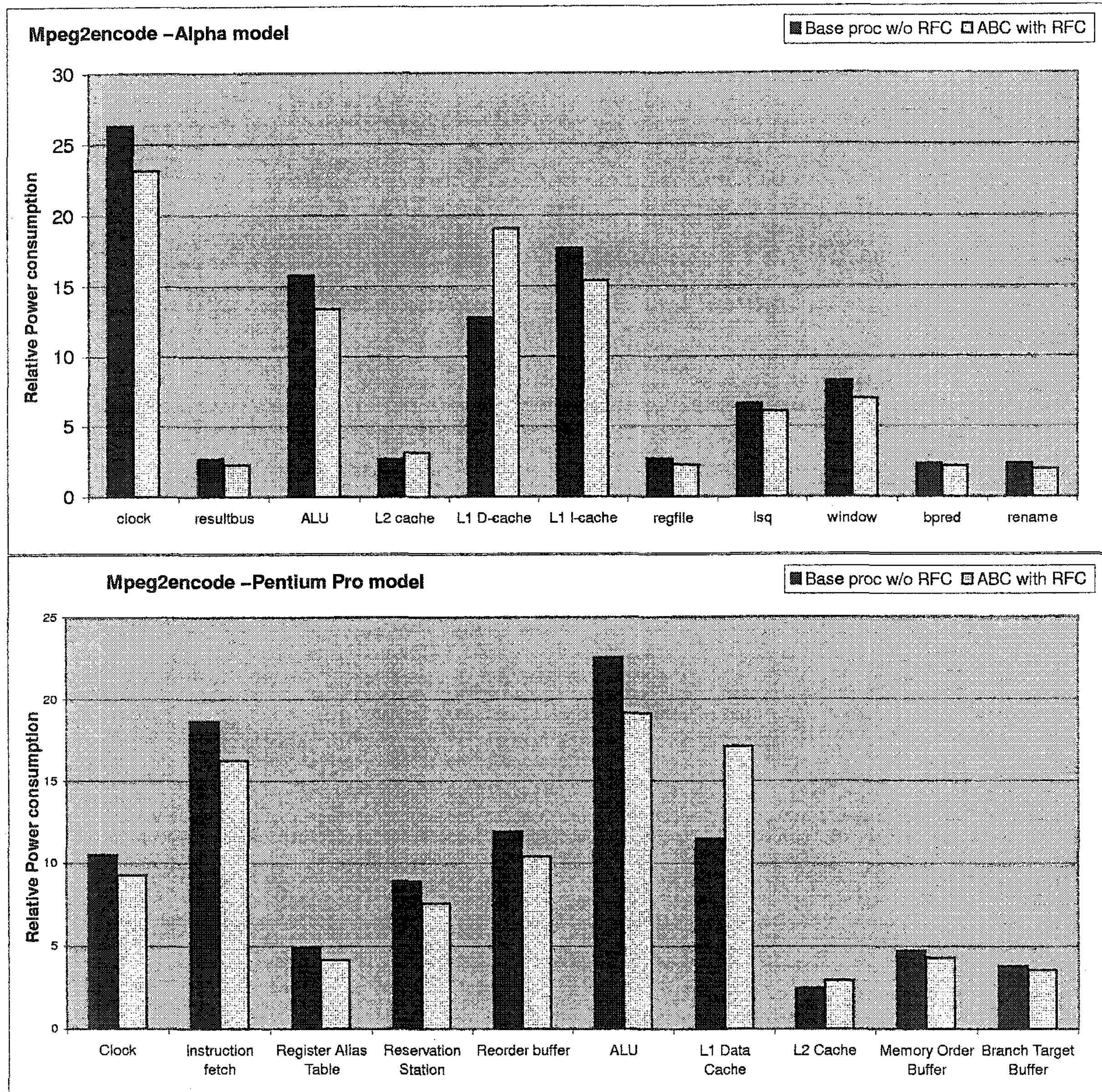


Figure 4.11 Component power utilization in ABC processor vs. base processor for MPEG2 encode application.

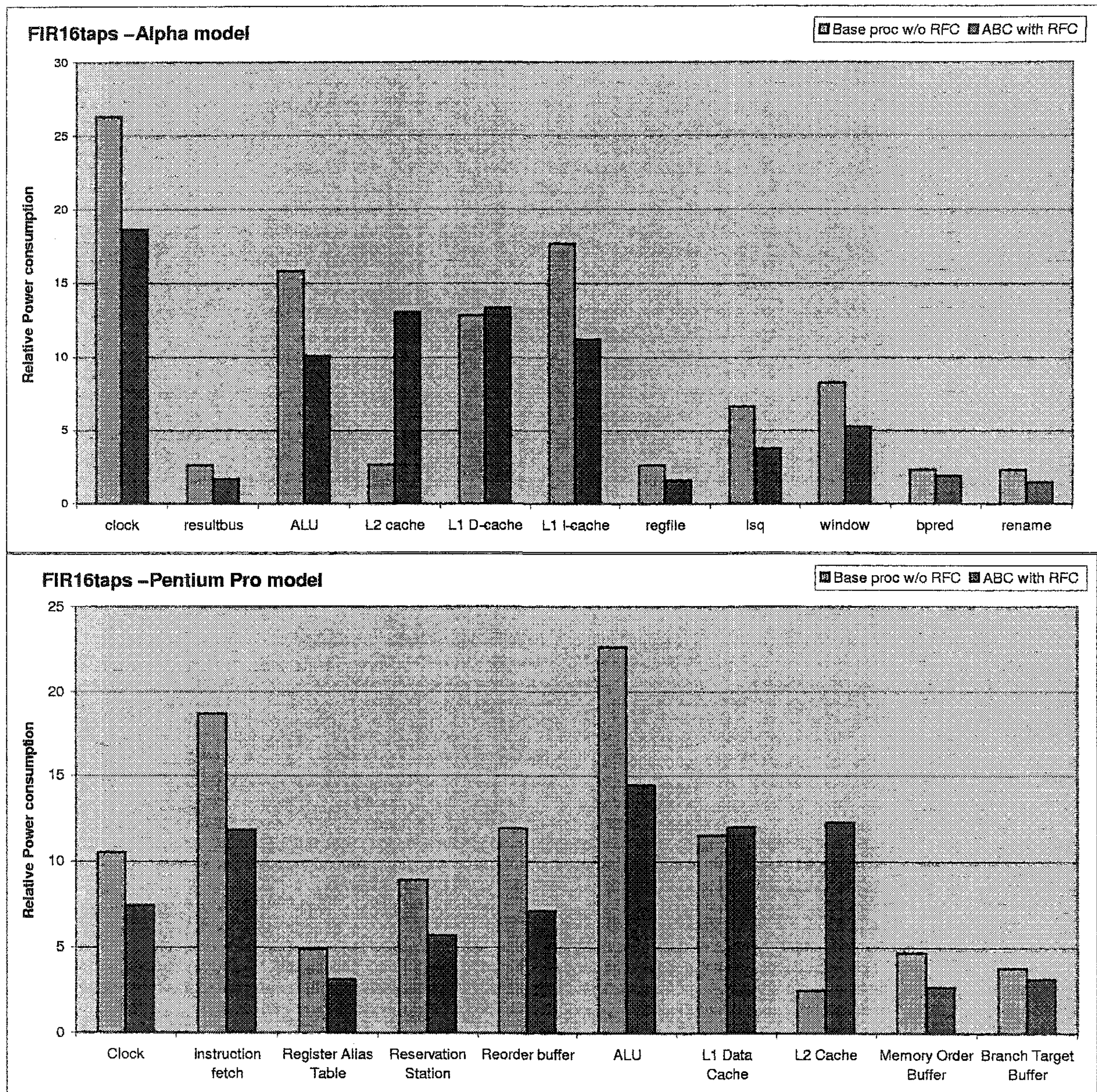


Figure 4.12 Component power utilization in ABC processor vs. base processor for FIR application.

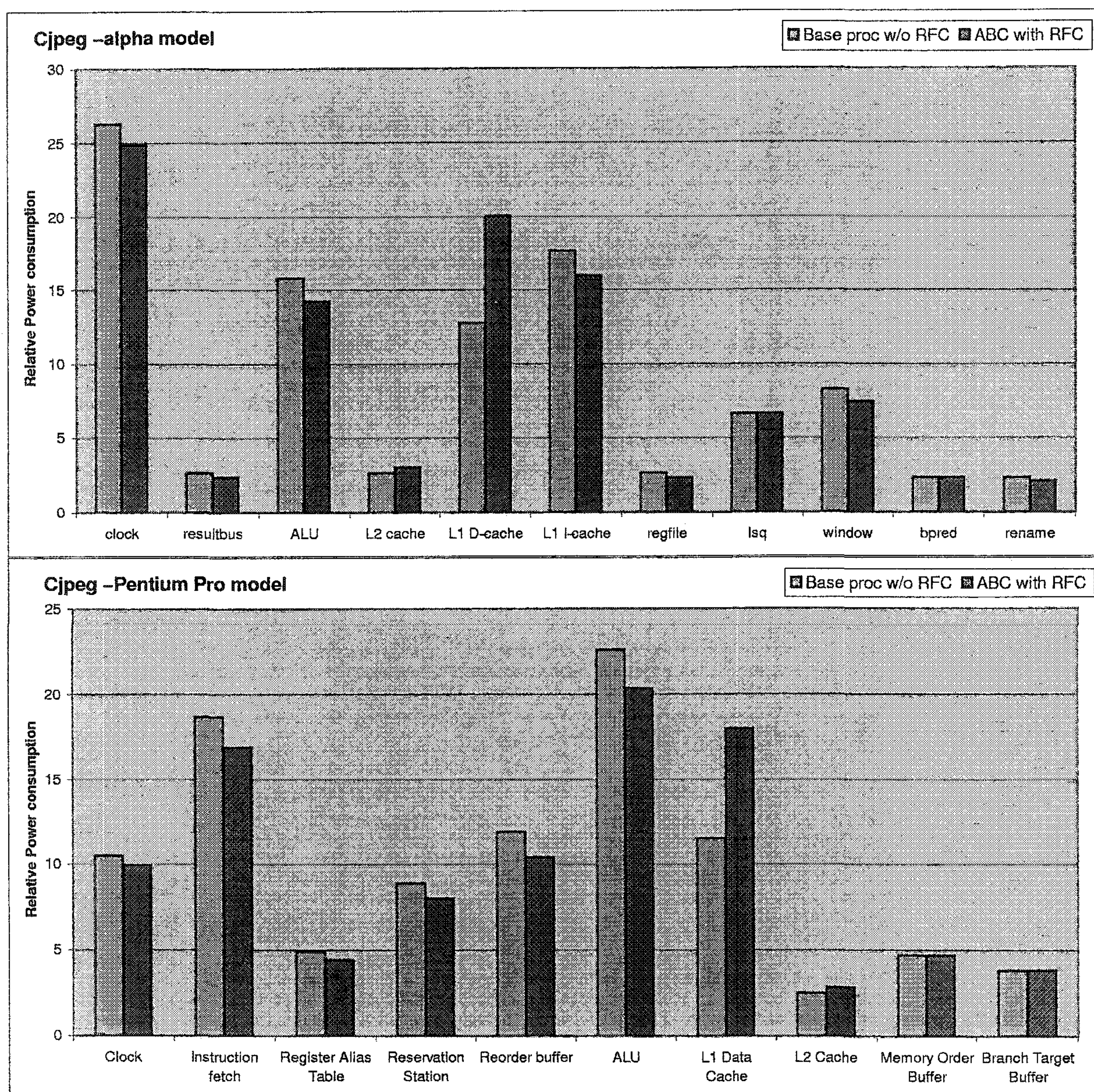


Figure 4.13 Component power utilization in ABC processor vs. base processor for cjpeg application.

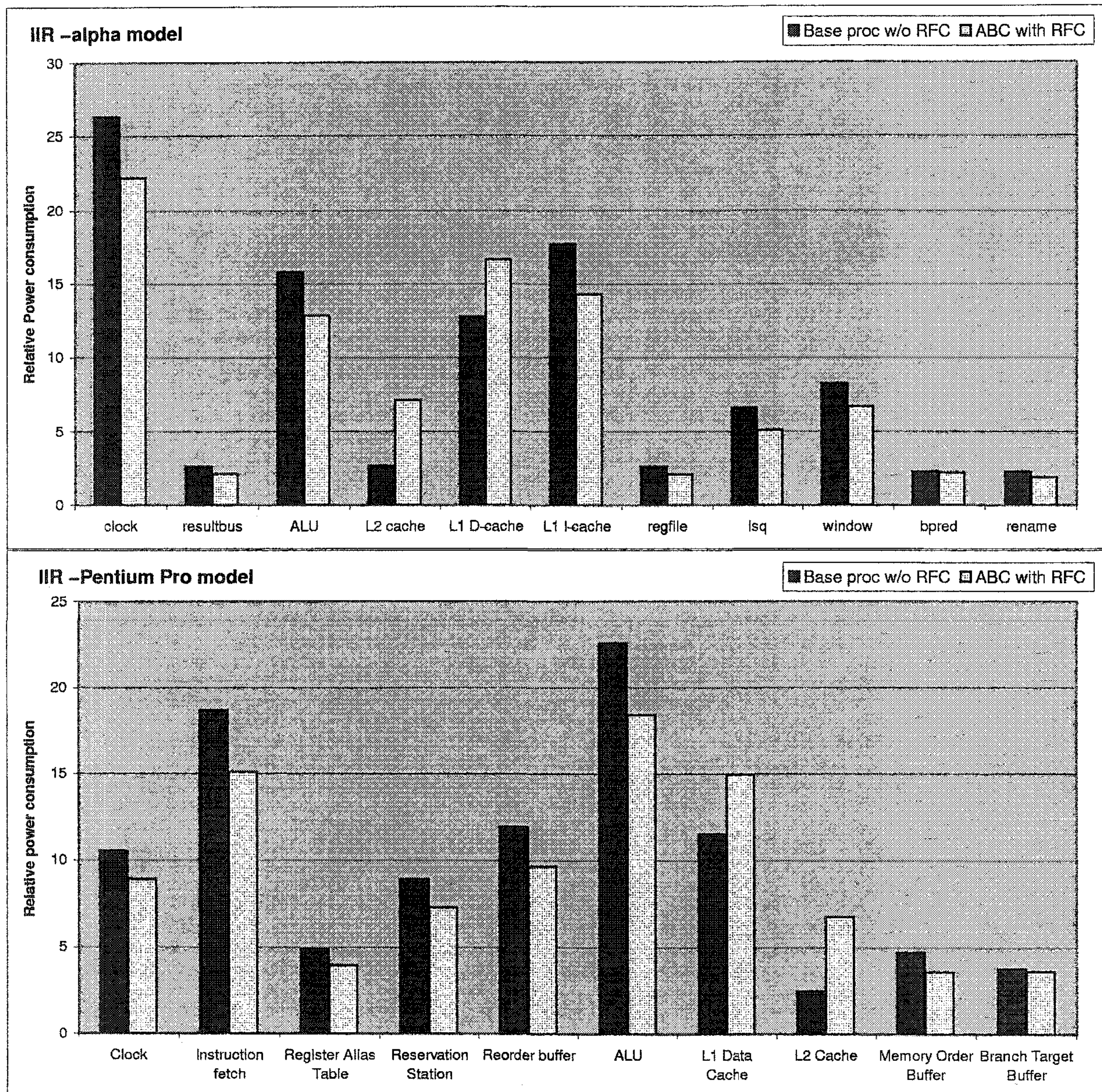


Figure 4.14 Component power utilization in ABC processor vs. base processor for IIR application.

despite a larger amount of power dissipation per access in RFC as compared to base cache. This occurs due to the reduced power dissipation in all the other on-chip components of the processor due to their reduced activity. And, the most important fact is that the reduction in overall power consumption is achieved along with a higher performance in executing the media applications.

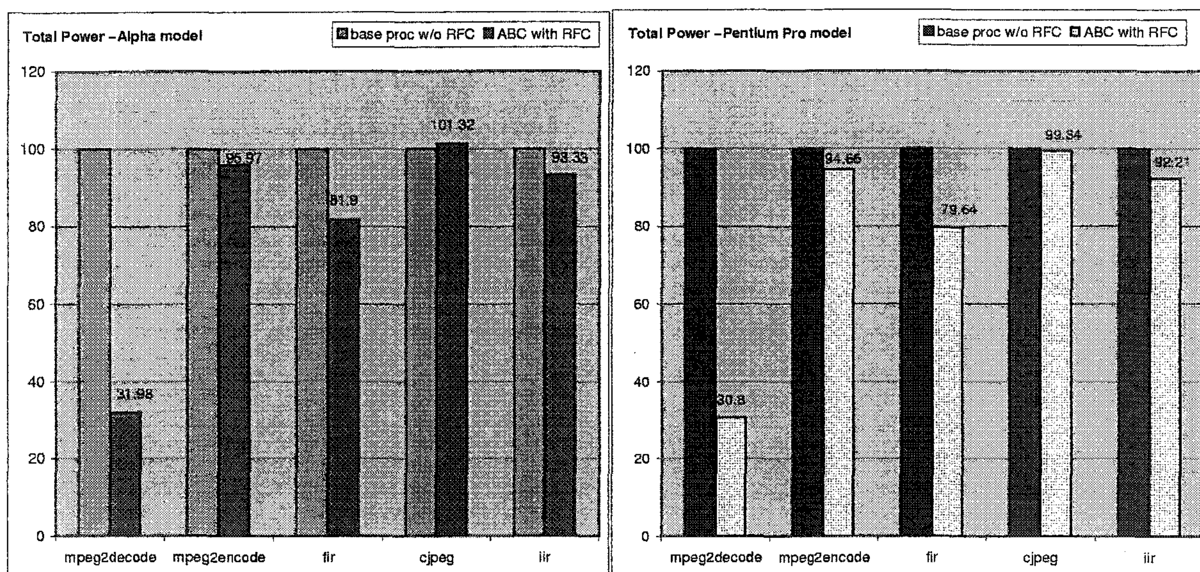


Figure 4.15 Total power utilization in ABC processor vs. base processor using (a) Alpha processor model (b) Pentium-Pro model.

4.5 Summary

The size of on-chip cache memory has been consistently increasing to keep up with the pace of enhancement in the processor technology and this trend is likely to continue even in deep-sub-micron technology. However, some applications may not be utilizing the full cache memory capacity and the performance improvement due to the increase of cache capacity may not be significant. In this chapter, we have given an overview of the development of Adaptive Balanced Computing (ABC) microarchitecture using a Reconfigurable Functional Cache (RFC). The RFC accelerates the computations using a specialized computing unit with minimal modification and overhead in time, power and area domains in the cache and microarchitecture.

The chapter discusses the high performance of the ABC processor in computing various multimedia applications. Besides, keeping in pace with the current requirement of power-aware architectures, it has been shown that the ABC processor delivers higher computing capacity while providing with significant savings in the energy dissipation in the various on-chip components of the processor. It can be argued that for the computations other than the media applications discussed in this chapter, and that will not be accelerated using the RFC, the RFC will still dissipate larger amount of power. However, as discussed earlier, it is not necessary to convert all the cache modules into computing units. Since the on-chip cache size is continuously increasing, it is possible to design some part of the cache as RFC modules and reserve them for computational purposes. They can be used as memory modules when an application demands larger cache capacity.

5 TIMING CONFIGURATION SWITCH

To build an ABC microprocessor, we integrate the reconfigurable functional caches (RFC) into a RISC superscalar microprocessor. A cache memory is partitioned into sub-cache blocks to implement the RFC in the existing cache structure. Some of these cache blocks can be configured as specialized computing units. This chapter explores two methodologies for the configuration of the RFC module. In the first scheme, the RFC is configured with the core function at the beginning of the multimedia application and this would result in a reduced cache capacity for the entire application. In the second scheme, the RFC is configured with the core function only when the core computation is necessary and the RFC module is released when not required to be used as a normal cache memory by the other computations within the multimedia application. With various cache mapping organizations, we study the overall impact on the performance of selected benchmarks, such as multimedia and DSP applications. The results prove that performance of each scheme varies depending on the structure of the application used. These issues are addressed as a part of our analysis on the ABC architecture.

The rest of the chapter is organized as follows. In Section II, we describe the first version of the ABC microprocessor. Section III presents an overview of the preliminary results obtained to support the case of building an ABC microprocessor. In Section IV, we propose various schemes to configure the core function into the RFC. Section V presents a comprehensive discussion on the impact of various parameters, related to the microarchitecture as well as the application, on the overall execution time. In Section VI, we discuss the results obtained from applying various RFC configuration schemes. Section VII presents the layout for the future work and finally, Section VIII concludes the discussion.

5.1 RFC Integrated Microarchitecture

The ABC architecture is built by incorporating a multiple-way set associative data cache memory in a RISC superscalar microprocessor. Each module in the set associative data cache is built as an RFC. One possible configuration of a ABC microprocessor with a 4-way RFC is shown in Figure 4.1. Each RFC module can be configured to a specialized computing function or can be used as a normal data cache memory module. The RFC module is constructed with multibit output Lookup Tables (LUTs). Two possible cache organizations with the address mapping including RFCs are shown in Figure 5.1. The organization of a reconfigurable cache module (RFC) had been extensively discussed in [41].

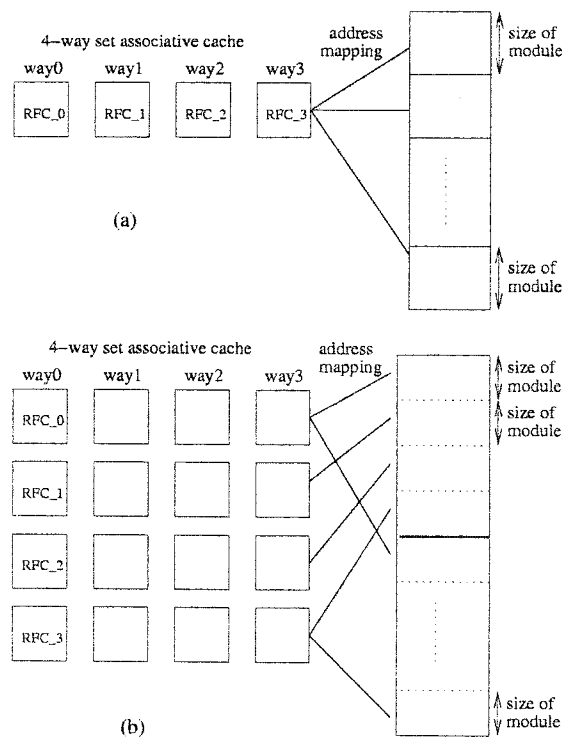


Figure 5.1 Reconfigurable Functional Cache (RFC) organizations and address mapping with (a) 4 cache modules (b) 16 cache modules.

To prevent the memory address space from becoming non-cacheable, not all of the

cache modules are configured as computing units at the same time, for the cache organization shown in Figure 5.1(a). RFCs can be implemented with a minimal cache modification in this organization. In addition, one module can be easily excluded from the cache operation because the cache partition is already provided in a multiple-way set associative cache memory. The cache memory capacity is reduced when an RFC converts into a computing unit. This results in a full dynamic associativity of cache memory when configuring. For example, if one out of four cache modules is configured to a computing unit, only 3-way blocks in all sets are left to map the address space in lower-level memory..

A further partition of cache memory within a module (way) is shown in Figure 5.1(b). To partition a large cache memory, we apply a similar address decoding mechanism in each cache module for an RFC. The decoder for each module is divided further to make a hierarchical decoding in higher address bits. This provides sixteen smaller-sized cache modules which can be built as the RFCs. The size of each module is based on the minimal size of RFC - 8KB shown in [41]. In this organization, when an RFC converts into a computing unit, the sets containing the RFC are less-cacheable (low associativity compared to other sets) while the other sets retain the same caching capacity. This scheme retains more storage than the full dynamic associativity organization of Figure 5.1(a) (each way corresponding to RFC) by converting a smaller portion of cache memory.

5.1.1 Instructions to Utilize RFC

There are three steps involved to perform a computation on RFC – configuration, loading input, and storing output. All these steps require only load/store class of instructions. To perform these operations, we add new instructions, named rfc load/store instructions. The format of the rfc instructions is the same as the conventional load/store instructions except for the target register field. We use the target register field for different purposes as shown in Figure 4.7. Three types of instructions, rfc_load_conf, rfc_load_in, rfc_store_out, and initialize/terminate instructions are added. The detailed format of the new class is described in Figure 4.7. We assume that only four

RFCs are available in the architecture. Individual instruction operations are explained briefly later in Section 5.1.2. Note that the FID field chooses the module that will be configured for a specific function. In this chapter, we use word-data type rfc instructions for the description purpose. For different types of data (for example, half-word or byte, etc.), the same instruction format can be used. Also, note that $rfc_lw_conf_end$ and $rfc_partial_set$ also performs a loading operation in addition to setting the mode. The $rfc_lw_conf_start$ instruction is used for a special setting in a RFC, such as use of multiple input buffers (i.e. input and intermediate data) as well as a hierarchical function identification of RFCs if more than four RFCs are implemented (e.g. function class followed by sub-FID).

5.1.2 Mechanism for the Computation in RFC

In out-of-order issue in a superscalar processor, any instruction which does not have a dependency with preceding instructions can be issued and executed at any time if the required resources are available. In addition, in a speculative execution, the next instruction stream in a code sequence can be executed speculatively. The out-of-order issue and execution may also happen among rfc instructions because there is no explicit dependency between rfc instructions. However, the $rfc_lw_in(1\ or\ 2)$ and rfc_sw_out instructions must not be issued and executed until the RFC has been configured. A speculative execution mechanism may issue the rfc instructions in any order. To avoid this type of exception, we add a special RFC state register. In the register, two bits are reserved for each RFC module. The two-bit RFC state information is organized as follows.

- 00 : NON-RFC/END-RFC - normal cache mode; no RFC performing a computation
- 01 : CONF - configuration mode for RFC as a computing unit
- 10 : CONF_DONE/START-RFC_EXE - end of configuration
- 11 : RFC_EXE - execution mode

The state transition is controlled by the *rfc* instructions as depicted in Figure 4.8. All the *rfc* instructions must access the RFC state register according to the FID field in the microcode and then check the current state with its CMD field. If it is an allowed state, the *rfc* instructions can be issued. Otherwise, the *rfc* instructions are stalled until the corresponding state is resolved.

5.1.2.1 Configuration

The configuration of RFC from a cache module simply implies loading all the contents of LUTs required to construct a computing unit. A normal cache read with a small modification directs specific data into the designated cache line. The required configuration data for RFCs resides in a reserved memory (address) space in main memory. The configuration is loaded into main memory when the system boots up. The *rfc_lw_conf_start* instruction sets the corresponding RFC state register. The subsequent *rfc_lw_conf* instructions load the configuration lines to the specific RFC without changing the RFC state.

Since the configuration data is held in main memory, the data accesses would be cache misses if the same configuration had not been loaded previously. This cache miss will replace the current clean/dirty lines in a write-back cache. Thus, we do not require any separate cache flushing. A simple modification of cache replacement mechanism, such as LRU (Least Recently Used), is required to replace the data in the specific cache module (RFC) with the loaded configuration data in a set-associative cache organization. The modified LRU scheme, which is set by *rfc_lw_conf_start*, replaces lines in the RFC only if *rfc_lw_conf* accesses in *CONF* mode. The configuration data in RFC should not be modified by other load/store instructions except the *rfc_lw_conf* instructions if the RFC state is not '00'. An additional operation in the modified LRU mechanism protects the configuration data by removing the lines in RFC from the replacement line list in LRU when conventional load/store instructions access the cache. Thus, the proposed LRU scheme consists of two operations, one for *rfc_lw_conf* to replace data in the specific RFC and the other for other load/store instructions (including *rfc_lw_in*1/2)

to access the rest of cache memory. Using the proposed LRU modification, the set associativity of cache memory is changed dynamically depending upon the use of RFC. A write operation to RFC is prevented by disabling the write enable in the RFC during the execution mode.

5.1.2.2 Execution stage

The new instructions are decoded in dispatch stage according to their specification described in Section 5.1.1. The $rfc_lw_in1/2$ instruction, which loads input data to an RFC computing unit, is decoupled from LSQ (Load Store Queue) and queued into an input buffer (IBUF) dedicated to each RFC. The decoupled IBUF provides data in-order to the RFC computing unit. In addition, the rfc load/store instructions process the input/output data independently in the separate buffers. Otherwise, these instructions pass/receive data to/from RFC serially in one buffer. The effective address for the $rfc_lw_in1/2$ instruction is calculated using the existing datapath and passed to the corresponding instruction in IBUF, not in LSQ. The details of mechanism to support out-of-order update (load) of input data in IBUF are discussed extensively in [42]. If no slot in IBUF is available, the following instructions including conventional instructions fetched from memory are stalled until IBUF is again available. Otherwise, it may require a complex mechanism for a decoupled fetch queue between rfc load instructions and other instructions. By queuing the $rfc_lw_in1/2$ instructions into IBUF in-order, the input data to be processed is provided to RFC in a correct order. This is like a reorder buffer mechanism for input data of the RFC unit to remove the impact of out-of-order execution. The input data from memory to the IBUF can be supplied out-of-order as in the conventional LSQ. Note that instead of providing IBUF with the data in write-back stage, the data may be directly loaded into the matched address slot in IBUF from data buses.

A whole function in an application may not be mapped to an RFC as a computing unit at one time. For instance, in an FIR filter, if the number of taps for the filtering coefficients is larger than the number of physical taps implemented in an RFC, we

configure the first set of taps in the RFC and then reconfigure it partially for the next set of coefficients at run-time. This can be achieved using a cache write operation in *rfc_lw_conf*. The *rfc_partial_set* instruction shown in Section 5.1.1 sets the RFC state register and all the required flags as *rfc_lw_conf_start* does. This mechanism protects the current configuration data to be retained in the RFC for the partial reconfiguration by setting the RFC state from *RFC_EXE* to *CONF* mode directly. The following *rfc_lw_conf* instructions reconfigure the designated RFC as done for the initial configuration. In this case, an input data stream may be processed by a fixed number of taps and partial results are stored. In the second iteration, input and partial results are loaded (using *rfc_lw_in*1 and *rfc_lw_in*2, respectively) and processed together. Thus, we provide two IBUFs. The computation in RFC is processed when both data elements are available.

After a computation is completed in the RFC, the output data is queued into an output buffer (OBUF). The OBUF is a simple FIFO register file since the queued data is already in-order. The output data in OBUF is stored into memory by the *rfc* store instructions. The *rfc_terminate* instruction sets the RFC state into the non-RFC mode after finishing an entire computation. This setting should be done in commit stage to avoid mis-execution of pending *rfc* instructions. If the same computation with the current configuration is performed in the near future, the RFC state (*RFC_EXE*) is not changed. The *rfc_lw_in*1/2 instructions do not affect any state when a misprediction/speculation or an exception occurs because the *rfc_lw_in*1/2 instructions do not modify the precise state in microarchitecture. All the repair to be done in this case is to flush the instructions in IBUF as done in a conventional LSQ.

5.2 Performance of ABC Architecture

For the simulation purposes, the design of the ABC processor is done using the Simplescalar tool set [9]. The source code of the simplescalar tool set is modified to incorporate the RFC design and the microarchitecture is modified accordingly to enable the communication between the main processor and the RFC. To analyze the perfor-

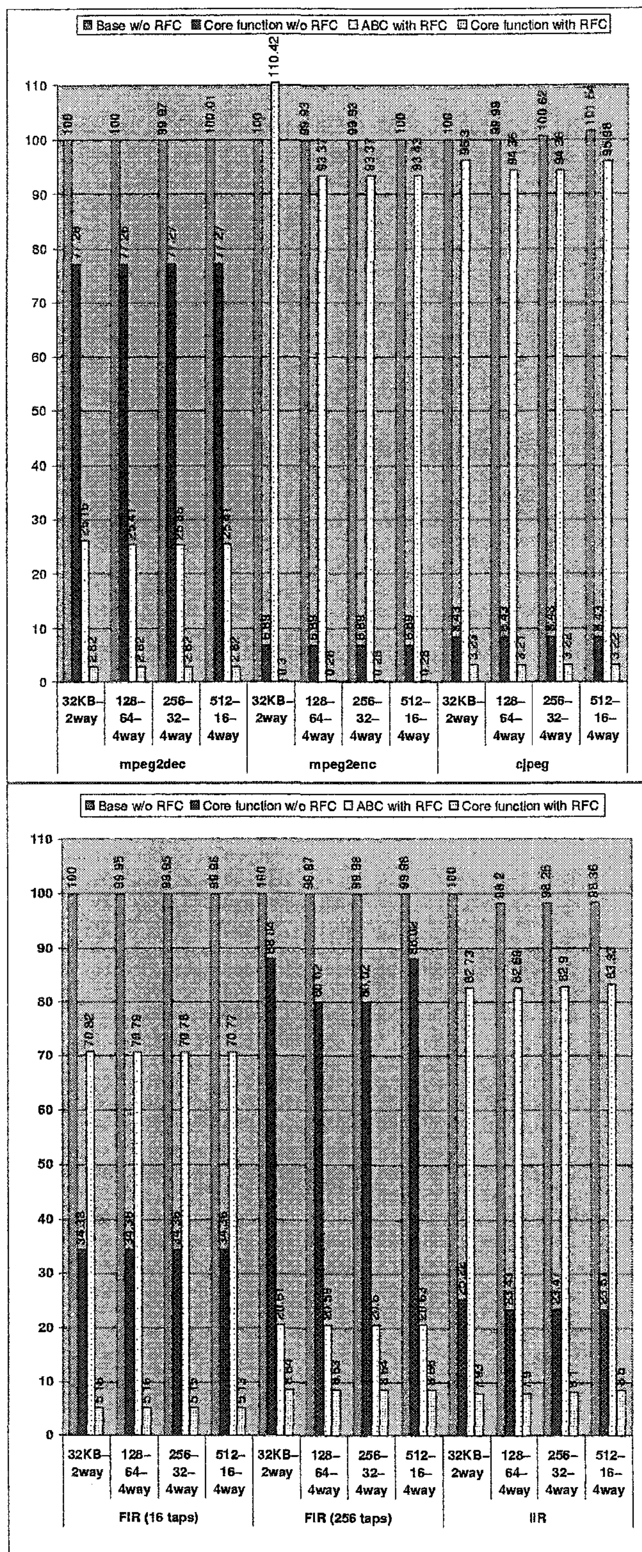


Figure 5.2 Normalized execution cycles in base processor without RFC and ABC processor with RFC, with varying cache organizations.

mance of the ABC architecture, the comparison is made between the number of cycles taken to execute each application with varying cache parameters. The performance, in terms of the total number of execution cycles normalized to the execution cycles in the base processor with 32KB 2-way set associative cache for each of the benchmarks [48, 36], mpeg2dec, mpeg2enc, cjpeg, FIR-16taps, FIR-256taps, and IIR, is shown in Figure 5.2. In Figure 5.2, each simulation result data set, for a particular cache structure, consists of four values: (a) the total application execution time in the general purpose processor (GPP) without the RFC, (b) portion of time taken for computing the core function in the GPP, (c) total application execution time in the ABC processor integrated with the RFC, and (d) the portion of time taken to execute core function in the RFC. The variation in the cache parameters is depicted as N-L-way, where N is the number of lines in the cache block, L is the line size in bytes and way represents the number of cache modules. It can be observed that all the varying cache organizations have been built in a 32-KB size cache.

The performance improvement can be obtained from the normalized total cycles for the execution of the overall function and also the core function. The specialized computing units configured in the RFCs improve the performance of each core function significantly. The most important factors for speed-up are the reduced number of instructions and the acceleration of computation with a specialized unit. Hence, it is observed that the overall speed-up is largely proportional to the frequency of the core function calls in the entire application. However, the initial simulation results show that the RFC is not a good idea to implement in a low associative cache in certain cases where it dramatically reduces the cache capacity. For example, a performance degradation is observed in the execution of mpeg2encode application in the ABC processor with 32KB 2-way set associative cache. This is due to the fact that in a 2-way set associative cache, when one of the cache modules is used as an RFC, the cache memory capacity is reduced to 50% and also the data cache acts as direct-mapped and hence causes the performance degradation. This motivated us to look further into this issue in the design of the ABC architecture, and hence we set-out to explore various methodologies of the management

of RFC between the functional unit and the cache memory modes.

5.3 RFC Configuration Schemes

The simulation shows that the base processor with a smaller size cache memory and integrated with RFC performs better as compared to a processor with larger cache size and without RFC. This may suggest a microarchitecture with a smaller cache size matching the performance of a traditional microarchitecture with a larger cache size. This is true in media applications due to the streaming nature of data and the lack of temporal locality. However, with a smaller cache memory size, the inadequacy would be that when the RFC is configured with the core function, the cache capacity for the other computations in the application would be reduced further. The direct-mapped cache for mpeg2encode increases the number of execution cycles by about 16.1% for 16KB and 7.2% for 32KB due to the significant increase in cache misses [42]. Thus the performance degradation in using RFC is caused by the reduction in cache capacity to half and in associativity to direct-mapped, when the RFC is configured with DCT/IDCT during the running of the mpeg2encode application.

In the multimedia applications, it has been observed that the applications like mpeg2decode, mpeg2encode, FIR and other filters are highly iterative and hence multiple instances of the core functions would be occurring during the processing of the entire application. For example, it has been observed that the DCT/IDCT function is called 129,600 and 8,448 times, during the running of mpeg2decode and mpeg2encode applications, respectively. Further, it is noted that the core functions like DCT/IDCT occupy a varying degree of percentage of entire application as shown in Figure 5.3.

The other factor that has a predominant effect on the utilization of the RFC for the execution of the core function would be the configuration overhead. It is the time, in number of cycles, taken to load the configuration data into the target hardware. The configuration overhead would be greater as more number of instances of RFC configuration occur. The configuration data to program a cache into a function unit may be either available in an on-chip cache or an off-chip memory. Load time for the configu-

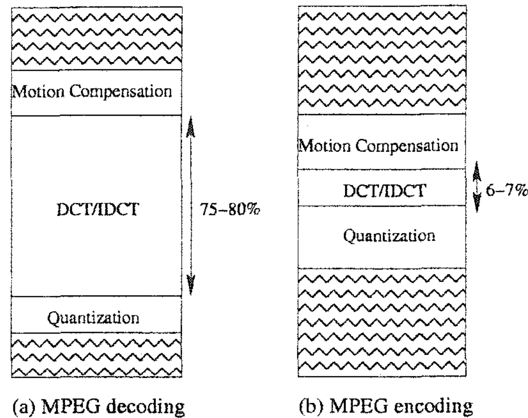


Figure 5.3 Distribution of core functions in MPEG applications.

ration data in the latter case will be larger than in the former case. The configuration data may be pre-fetched by the host processor from the off-chip memory to reduce the overall configuration overhead. In the ABC architecture, the time to configure the RFC, from the normal cache memory mode to a functional unit mode, depends on the number of cycles to write words into a cache. On the other hand, an RFC operating as a functional unit can also be partially configured at run-time using write operations to the cache. When a partial reconfiguration occurs, the function unit must wait for the reconfiguration to complete before feeding the input data. Since the computation data (input and output data) and the configuration data (contents of LUTs) for an RFC unit share the global lines for data buses, we cannot perform both the computation and the partial reconfiguration simultaneously. It is possible to perform both the operations simultaneously, if separate data buses for computation and configuration are provided, which would dramatically increase the cost of the microarchitecture design.

Thus it becomes imperative for us to study and devise various RFC management schemes for an efficient utilization of the computing and memory resources to achieve a better performance from the integration of the RFCs into a conventional microprocessor.

5.3.1 Scheme1: One-time configuration of RFC

In the current design [42] of the ABC architecture, the configuration of the core function into the RFC module is performed a priori, i.e., at the beginning of the application so that the configuration of the same core function at multiple instances can be avoided. However, due to this method, the computations other than the core function and those computed by the main processor, are denied the usage of all the cache modules. Hence, as discussed earlier, for an entire MPEG application in an architecture with 32KB 2-way cache, only 50% of cache memory is available for usage, besides setting the cache associativity to direct-mapped, even when the core function computing RFC module is inactive.

This scheme would prove to be beneficial when the core function is dominant in its execution over the entire application as in the case of DCT/IDCT function in the mpeg2decode. However, the scheme has some limitations for adoption, as it can be observed that it will have a negative impact when the percentage of core function is significantly small over the entire application as in the case of DCT/IDCT function in mpeg2encode application. Also, the scheme would not be better for adoption, when the configuration overhead, the total number of cycles taken to configure the core function, would prove to be significantly smaller and can be offset by the maximum utilization of the cache memory resources.

5.3.2 Scheme2: Continual configuration of RFC

To overcome the limitations of the continuous denial of usage of full cache memory capacity for computations other than the core function, the RFC module can be configured with the core function at every instance of its occurrence and then be released to function as normal cache memory for the benefit of other computations. Thus the RFC module has to follow a pattern of continuous switching of modes, between the normal cache memory mode and the functional unit mode.

As it was originally anticipated and as argued in the case of the first scheme, the proposed continual reconfiguration scheme also has its gains and limitations. This scheme

will have a negative impact when the percentage of core function execution time is significantly large in the entire application as in the case DCT/IDCT function in the mpeg2decode. Besides, the scheme would prove to be limiting when the number of instances of the occurrence of the core function is high as the configuration overhead would be significantly large.

5.4 Analysis of Execution Time

In this Section, we present our arguments to justify the need of an analytical study of the two RFC configuration schemes described above. To adopt the RFC based ABC architecture for processing various multimedia applications, one of the above schemes has to be incorporated in the design. In view of the gains and limitations of the proposed RFC configuration schemes, and keeping in view of the nature of the multimedia applications where the percentage of core function in the entire application and the number of instances of occurrence of core function vary, we develop a mathematical model for the analysis of the performance in terms of total execution time, to determine which scheme proves to be beneficial under varying circumstances.

As a precursory, before presenting the mathematical model, the discussion on the observations, from which the model has been developed, would be appropriate. From the preliminary results of simulation, it is observed that the performance of RFC based ABC architecture, in the processing of various multimedia applications, is almost the same irrespective of various cache designs like 32KB 2-way, 64KB 2-way, 64KB 4-way and 128KB 4-way. Hence to minimize the design cost, let us assume that a cache structure of 32KB 2-way is available. In that case, only 50% of cache memory would be available for the entire application, when the one-time RFC configuration scheme is incorporated into the architecture design. Hence it needs to be studied whether using the RFC module for the configuration of the core function, only when required, would be advantageous or not.

Further, the architecture and application parameters, that will have significant impact on the overall execution time, have to be considered for the trade-off analysis among

various schemes. The parameters that need to be considered are, the number of cycles required for configuration of RFC (C_p), the number of instances of occurrence of core function in the application (N), the fraction of time the core function is processed over the entire application (P), and the cache blocking factor (ϕ). Note that (1-P) represents the fraction of time for the computations other than the core function. Also, since we design the architecture for the faster execution of applications by accelerating the core function, we assume that there always exists a portion of the core function that can be mapped into the RFC. Hence the assumption that $P \neq 0$, holds true for the entire discussion.

Table 5.1 Architecture Design Parameters.

C_p	Cycle time for RFC configuration when data is fetched from off-chip memory.
P	Fraction of core function over the entire application.
N	Number of instances of core function over the entire application.
ϕ	Cache blocking factor.
X_A	Execution time of core function in the base processor.
X_B	Execution time of core function in the RFC functional unit.
S	Speed-up of core function = $(\frac{X_A}{X_B})$

The total execution time (X_{org}) of an application in the base processor without RFC can be represented by the expression:

$$X_{org} = X_A \left(\tfrac{1-P}{P}\right) + X_A = \frac{X_A}{P} \tag{5.1}$$

Similarly, the total execution time (X_{rfc}) of the application in the RFC integrated ABC processor can be represented by:

$$X_{rfc} = X_A \left(\tfrac{1-P}{P}\right) \phi + C_p' n_c + X_B \tag{5.2}$$

The $X_A(\frac{1-P}{P})$ accounts for the execution time of the computations other than the core function. When the continual RFC configuration scheme is employed, the cache blocking factor (ϕ) ≈ 1, i.e., all the cache modules are available for utilization as cache memory during the execution of computations other than the core function. The ϕ is assumed to be approximately equal to 1 instead of being exactly equal to 1 in the continual configuration scheme, due to the fact that whenever the RFC module is released to be used as a cache memory module, the data is written into the cache module to be readily used by the main processor and that data is again replaced by the configuration data when RFC is switched to functional unit mode. Thus, in the process of RFC being switched between various modes, the main processor will be experiencing cache misses, which might be hit under normal cache operation. When the one-time RFC configuration scheme is employed, the cache blocking factor (ϕ) > 1, as one of the cache modules is reserved for the functional unit and hence execution of the computations other than core function would take more time due to the rise in cache miss rate and the reduced cache capacity.

The $C'_p\, n_c$ accounts for the RFC configuration overhead over the entire application. When the one-time RFC configuration scheme is integrated into the design, the configuration overhead will be significantly smaller, as $n_c = 1$. Also, $C'_p = C_p$ since we assume that during the first time configuration of RFC, all the configuration data would be a miss in the cache and hence need to be fetched from the off-chip memory. However, when the continual RFC configuration scheme is employed, the configuration overhead forms a considerable portion of the total application execution time, as $n_c =$ N. However the bright spot in this case is that the simulation has proved that $C'_p \ll C_p$, since the subsequent instances of configuration of RFC with the same function as the preceding instance configuration will not take as many cycles as the first instance of configuration. This is due to the fact that, between the adjacent calls of the core function when the other computations are being executed, a few data blocks in the RFC module get replaced and hence during the subsequent configuration, only the corrupted configuration data is fetched from the off-chip memory.

Table 5.2 Variation in parameters for various RFC configuration schemes.

	One-time configuration	Continual configuration
Cache blocking	$\phi > 1$	$\phi \approx 1$
Configuration instances	$n_c = 1$	$n_c = \mathrm{N}$
Configuration cycles	$C'_p = C_p$	$C'_p \ll C_p$

The summary of the variation of parameters with respect to each of the schemes employed is as shown in Table 5.2. The parameters, ϕ in the case of one-time RFC configuration scheme, and C'_p in the case of continual RFC configuration scheme are the two factors that affect the total execution time predominantly. It is not possible to exactly quantify these two factors, as both ϕ and C'_p are entirely dependent on the amount of data cache accesses, the cache organization and the cache miss rate. However, fortunately we can define the upper bounds for each of the parameters with the goal of obtaining a better performance from the RFC based ABC microprocessor. The related discussion is given in the following subsections.

5.4.1 Execution Time Analysis With One-Time RFC Configuration

When the design of ABC architecture is incorporated with the one-time RFC reconfiguration scheme, the expression for X_{rfc} will be modified, after substituting relevant parameters from Table 5.2, as given by the following expression:

$$X_{rfc} = X_A \left(\tfrac{1-P}{P}\right) \phi + C_p + X_B \tag{5.3}$$

We are aware that, in the worst case, the configuration time (C_p) can be in the order of thousands of cycles while, X_B would be in the order of millions of cycles and hence C_p can be conveniently neglected in the expression. Now, for a gain in the performance, the execution time of application in the RFC based ABC processor should be less than the execution time of application in the base processor. Hence, solving $X_{rfc} < X_{org}$ yields

$$X_A \left(\frac{1-P}{P}\right) \phi + X_B < \frac{X_A}{P} \tag{5.4}$$

from which we can obtain the upper bound for the cache blocking factor as:

$$\phi < [\frac{1-(\frac{P}{S})}{1-P}] \tag{5.5}$$

From the above expression, it is obvious that the lower bound on ϕ is 1, as $\frac{X_B}{X_A} \leq 1$ under all circumstances, i.e., the speed-up obtained for the core function by computing in the RFC would be at least 1.

The variation of the cache blocking factor with respect to the fraction of the core function (P) and the speed-up of the core function (S) is shown in Figures 5.4 and 5.5. It can be observed that, for applications with a larger fraction of the core

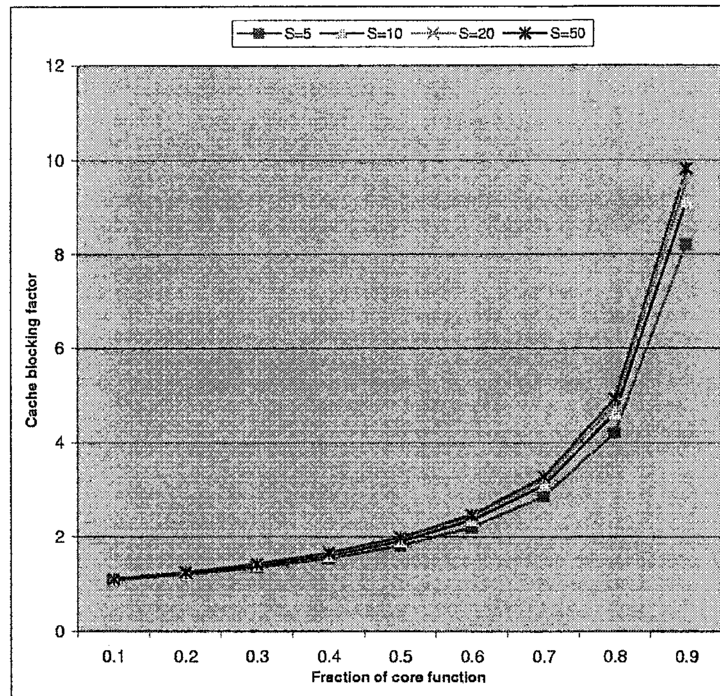


Figure 5.4 Variation of cache blocking factor with the fraction of core function.

function, the upper bound of the cache blocking factor is large which signifies that the cache blocking factor (ϕ) can be raised up to its upper bound without degradation in the overall performance when the one-time RFC configuration scheme is implemented. Similarly, when the percentage of the core function is small, the upper bound of ϕ is

small and hence the value of ϕ needs to be maintained within that tight bound to avoid a degradation in the performance of the ABC processor. Besides, it can be observed that for a fixed portion of the core function, the upper bound of the ϕ remains almost constant irrespective of a large variation in the speed-up of the core function. Thus it can be deduced that for applications with smaller fractions of core function, it might become necessary to adopt the continual RFC configuration scheme where the impact of ϕ will be nullified, as discussed in the following subsection.

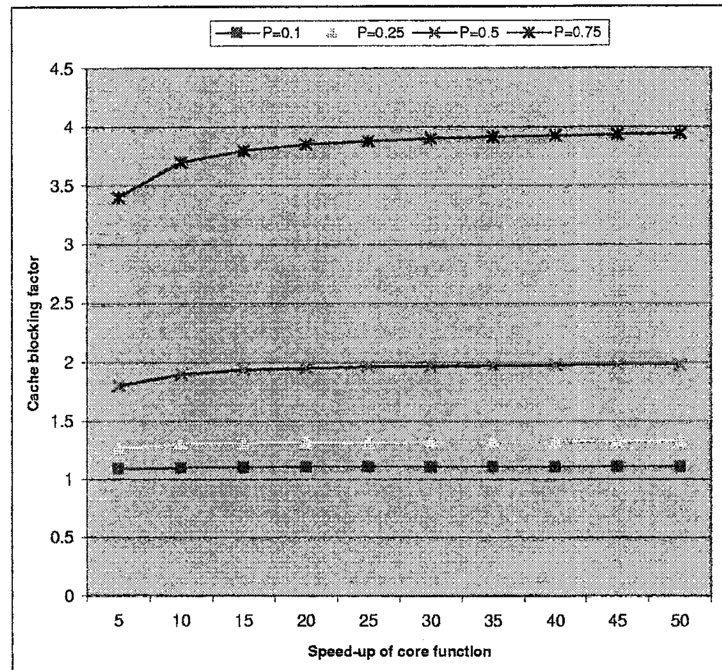


Figure 5.5 Variation of cache blocking factor with speed-up in core function.

5.4.2 Execution Time Analysis With Continual RFC Configuration

When the design of ABC architecture is integrated with the continual RFC reconfiguration scheme, the expression for X_{rfc} will be modified, again substituting the relevant parameters from Table 5.2, as shown:

$$X_{rfc} = X_A \left(\frac{1-P}{P}\right) + C'_p N + X_B \tag{5.6}$$

As argued in the earlier subsection, for the implementation of continual RFC configuration scheme, it is assumed that the cache blocking factor (ϕ) = 1 for practical purposes, though in the ideal scenario, $\phi \approx 1$. Now, for a gain in the performance, the execution time of application in the RFC based ABC processor should be less than the execution time of application in the base processor. Hence, solving $X_{rfc} < X_{org}$ yields

$$X_A \left(\frac{1-P}{P}\right) + C_p' N + X_B < \frac{X_A}{P} \tag{5.7}$$

from which the upper bound for the average configuration time (C_p') can be obtained as:

$$C_p' < \left[\frac{X_A - X_B}{N}\right] \tag{5.8}$$

The upper bound on the average configuration time (C_p') signifies that the total configuration overhead in the application should not exceed the difference in the execution time of the core function in the base processor and the RFC. This inference synchronizes with the generalized condition for the RFC based ABC processor to perform better than the base processor.

5.4.3 Effect of Number of Core Function Instances (N)

From the expression for X_{rfc}, it can be observed that, in general, the execution time $\propto$ N, i.e., more the number of instances of the core function in the application, more will be the execution time due to the higher configuration overhead. However, for a considerably smaller number of N, the continual RFC configuration scheme would prove to be beneficial while the effect of a larger N can be offset by employing the one-time RFC configuration scheme. For example, the simulation study shows that the DCT/IDCT function has been called 129600 times in the mpeg2decode application, while it has been called 8,448 times in mpeg2encode application. Hence, while running the mpeg2decode application, the one-time RFC configuration need to be employed while the continual configuration scheme proves to be better while running the mpeg2encode application, as discussed in the results section.

5.4.4 Effect of Percentage of Core Function (P)

From the expression for X_{rfc}, it can be observed that the execution time $\propto (\frac{1-P}{P})$ and hence, more the percentage of the core function in the application, higher will be the speed-up of the overall application. This phenomenon is typical of the characteristic of Amdahl's Law, where the speed-up of the overall application is proportional to the portion of the application being accelerated. It can be deduced that when the portion of core function is smaller over the entire application, then it will not make a significant impact even when it is accelerated using an RFC. The simulation results, as shown in Section III, proves this principle. From Figure 5.2, it can be observed that the speed-up of around 4X is obtained in the execution of mpeg2decode application in RFC integrated ABC processor while the maximum speedup obtained in the execution of mpeg2encode application in RFC integrated ABC processor is only 1.07X and it can be seen that the DCT/IDCT computation occupies a portion of 77.27% in the mpeg2decode application, while it occupies a portion of only 6.89% in the mpeg2encode application.

5.4.5 Effect of Cache Blocking Factor (ϕ)

When all the cache modules are not available during the application execution time, size of the cache reduces along with reduction in the cache associativity, which causes the cache miss rate to increase. Subsequently, the execution time of the application increases. Hence, from the expression for X_{rfc}, it is obvious that the execution time $\propto \phi$. However, when the portion of the computations other than the core function in the overall application is significantly smaller, it will not be appropriate to employ the continual RFC configuration scheme, as the gain obtained due to the availability of full cache capacity for smaller execution time would be offset by the configuration overhead.

5.5 Results and Analysis

The normalized execution time obtained by running the mpeg2encode application in the ABC processor with different RFC configuration schemes and with varying cache

organizations, is compared with the base processor performance as shown in Figure 5.6. Besides, the relative improvement in the performance of ABC processor with two different RFC configuration schemes is shown in Figure 5.7.

From Figure 5.6, it can be observed that the performance of ABC processor with continual RFC configuration scheme is better than the base processor without the RFC and also the ABC processor with one-time RFC configuration scheme, for all the cache organizations in the execution of mpeg2encode application. This is due to the fact that the percentage of the core function over the entire application is significantly smaller and also the number of instances of the core function is not big enough to generate a considerable configuration overhead. Note that the portion of core function indicated in the continual RFC configuration scheme includes both, the RFC configuration overhead and the computation time for the core function. Another interesting observation is that when the number of times (N) the core function is called for configuration in RFC is 8,448 instances and the configuration time (C_p) for full loading of the RFC module with the configuration data from the off-chip memory is 798 cycles, the total configuration overhead is expected to be 6,741,504 cycles. However, due to the cache hit of configuration data for subsequent RFC configurations, the total configuration overhead is determined to be 2,215,425 cycles which is only 32% of the expected configuration overhead.

For the above reasons, the continual RFC configuration scheme performs better even in the 32KB 2-way cache while the one-time RFC configuration scheme results in a performance degradation, as shown in Figure 5.7. Also, it can be observed that there is an improvement in the performance of the continual RFC configuration scheme, with the reduction in the number of blocks in the RFC cache module. For example, it can be observed that the performance of continual RFC configuration scheme improves for the RFC cache module with 512, 256 and 128 blocks, in that order. This is due to the fact that, as the number of blocks in a cache module increases, the configuration time for RFC module increases and subsequently configuration overhead is significant, thus having a negative impact on the overall performance.

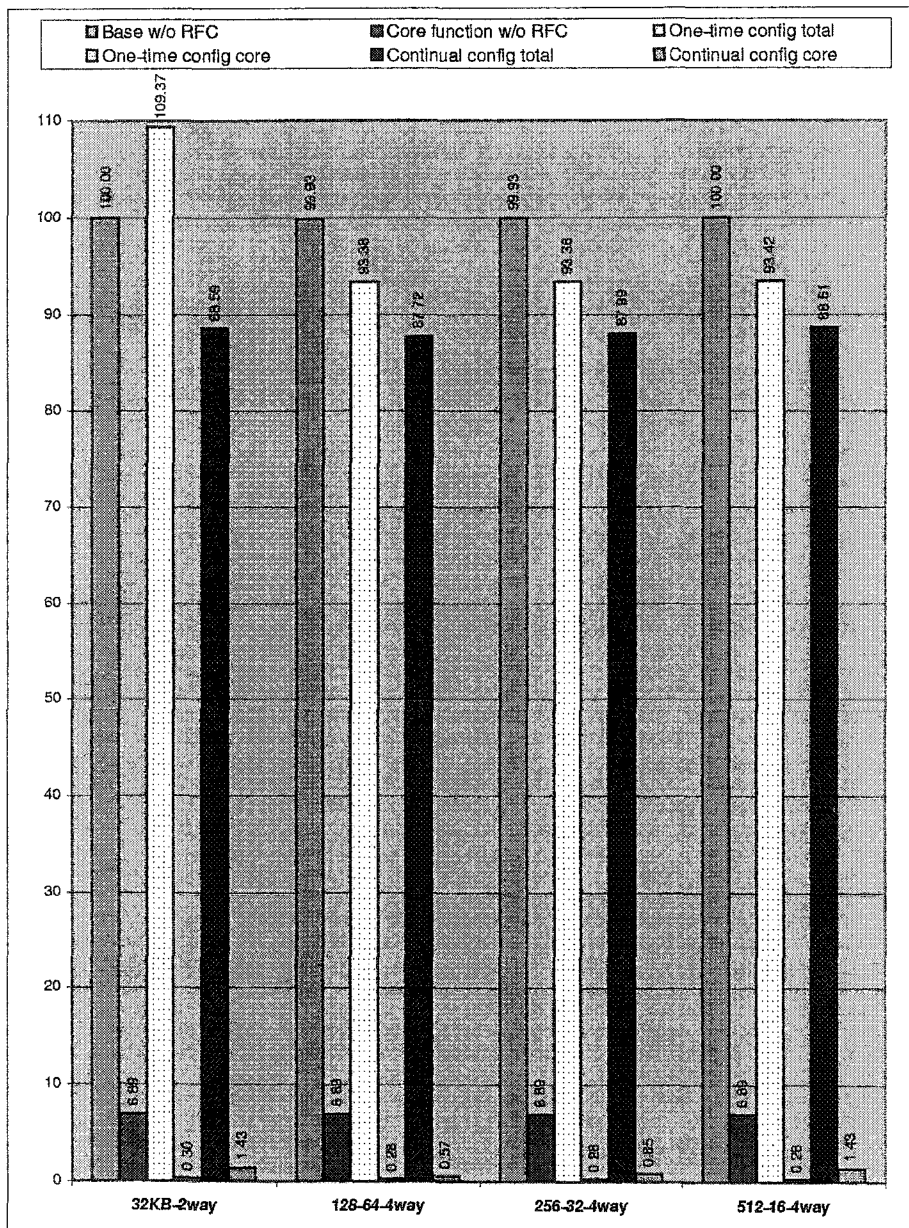


Figure 5.6 Normalized execution cycles in the base processor without RFC, and the ABC processor with different RFC configuration schemes.

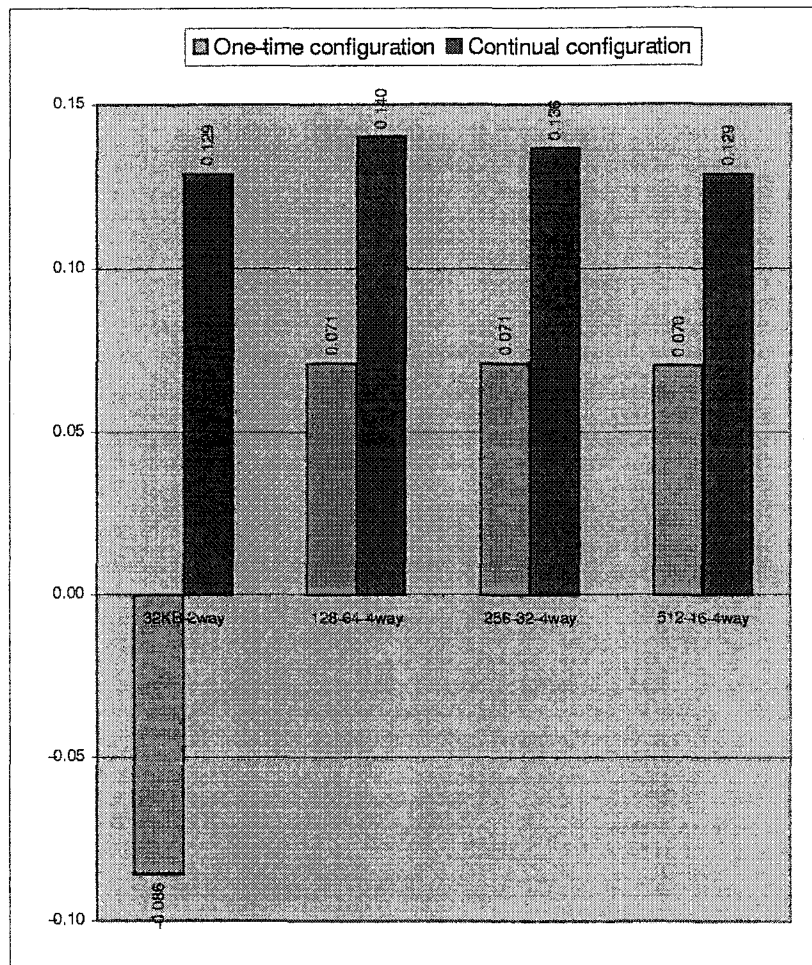


Figure 5.7 Relative performance improvement in two RFC configuration schemes.

5.6 Future Work

Conventional programmable processors, such as microprocessors, have support for large on-chip instruction caches. In the 1990's, the desktop processors adopted superscalar techniques [65] to exploit the instruction-level parallelism (ILP). Accordingly, to meet the demand for larger instruction depth, the on-chip instruction caches have been growing larger over time and can now hold hundreds to thousands of instructions. On the other side, the reconfigurable array has an on-chip space requirement of only one instruction per element, i.e., the single instruction which tells the logic array what function to perform and how to route its inputs and outputs. Increasing the number of on-chip instructions allows the device capacity to be used instantaneously for different operations at the cost of diluting the area used for active computation and hence decreasing device computational density [20]. Figure 5.8 shows where both traditional and reconfigurable organizations lie in the wide architectural space. When the reconfigurable array is active in computing the core function, the space required for storing a stream of instructions is minimal. This can be observed from the fact that, when mpeg2decode application is run on the preliminary design of ABC architecture, the total number of instructions executed when the core function is computed in RFC is only 23.9% of total instructions executed when the entire application is run in the microprocessor alone. Thus the benefits of using the instruction cache modules for computing purposes need to be explored.

Conceptually, there is no difference between the organization of instruction and data caches and the accesses in both the caches are associative. However, the caches differ in their number of read and write ports. An instruction cache needs one write-port, since a cache block is targeted to be filled from the memory. A data cache should have more than a few write ports depending on the number of instructions committed each cycle. The loading of the configuration data into the instruction cache from the memory is equivalent to the fetching of the instructions from the memory and thus do not require any modifications in the existing ABC architecture design. When the data cache is acting as a computing unit, the main processor makes the computation data flow

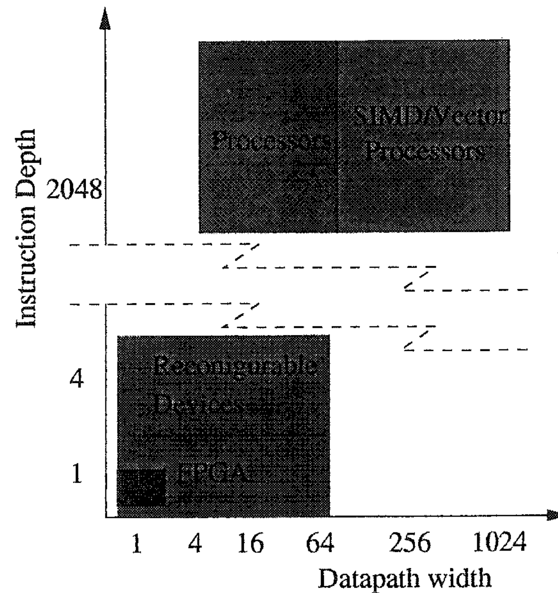


Figure 5.8 Architectural Design Space.

between the memory and RFC by using the normal load and store operations. The input data supplied for execution are diverted into an input buffer (IBUF) from where RFC consumes the data, and the results from RFC are stored in an output buffer (OBUF) from where the main processor reads the data to store back into memory. When the instruction cache is used as computing unit, a slight modification in the architecture is necessary to allow the processor to divert the data into the input buffer for the instruction cache module, and a mechanism need to be provided to enable the processor to read the results from the output buffer of the instruction cache module. A such mechanism to use the instruction cache modules as the computing units will give a chance to build a larger number of on-chip computing cache modules and thus a greater degree of computational power to the adaptive computing model.

5.7 Summary

We have given an overview of the development of Adaptive Balanced Computing (ABC) microarchitecture using a Reconfigurable Functional Cache (RFC) module. The RFC accelerates the computations using a specialized computing unit with minimal modification and overhead in area/time domains in the cache and microarchitecture. The summary of preliminary results are presented to support the cause of development of ABC microprocessor. Further, in continuance of our effort to build an efficient ABC architecture with improved performance, various RFC configuration schemes have been studied. A detailed study of the performance analysis of the architecture, in terms of the execution time of the application, is given. The impact of various architectural parameters and the factors governing the structure of an application over the execution time of an application has been extensively studied. With the help of the study undertaken, the design of ABC microprocessor can be incorporated with the dynamic decision capability, so that appropriate RFC configuration scheme is chosen dynamically for running a particular application.

6 IP ADDRESS LOOKUP ENGINE

In this chapter, we explore the application of reconfigurable architectures to the field of computer communications and networks. With a rapid increase in the data transmission link rates and an immense continuous growth in the Internet traffic, the demand for routers that perform IP forwarding at high speed and throughput is ever increasing. The key issue in the router performance is the IP address lookup mechanism based on the longest prefix matching scheme. Earlier work on fast IPv4 routing table lookup includes, software mechanisms based on tree traversal or binary search methods, and hardware schemes based on Content Addressable Memory (CAM), memory lookups and the CPU caching. These schemes depend on the memory access technology which limits their performance. The research proposes a Binary Decision Diagrams (BDDs) based optimized combinational logic for an efficient implementation of fast address lookup scheme in reconfigurable hardware.

6.1 Introduction

The effective handling of the tremendous amount of Internet traffic and its continuous doubling every few months depends on the efficacy of the routers. The key issue in router performance is the IP address lookup mechanism used for the ferrying of the large number of incoming communication packets to respective outgoing links. The router uses the destination IP address encoded in the incoming packet to lookup the next hop router to which the packet has to be forwarded. Since the introduction of Classless Inter Domain Routing (CIDR) in 1993, the IP address lookup mechanism has been designed based on the longest prefix matching (LPM) algorithm. The problem involves two steps, first

to search the routing database to obtain the longest matching prefix from all the possible prefixes that match the particular destination IP address, and second to retrieve the next hop port for the longest matched prefix. With the advent of optical medium for data transmission, the link rates have rapidly increased from 10 Mbps Ethernet to 40Gbps OC768c and there is every possibility of the line rates increasing well beyond. The primary concern in the design of next generation routers is to obtain maximum possible packet throughput to meet the demand from the high-speed transmission links. The continuous increase in the number of users on the Internet causes the creation of some explicit routes for certain users and constrains the router from aggregating the routing table effectively. This results in the expansion of routing tables and thus the search space of prefixes against which the destination address of each packet needs to be matched. Further, when the IPv6 routing protocol is introduced where the address length is 128 bits, the problem of routing millions of communication packets every second, based on longest prefix matching, becomes a labyrinth. In these circumstances, where IP routing tables are expanding in both the dimensions, i.e., address length and number of prefixes, the routing mechanisms developed should be capable of providing the throughput demand from high-speed transmission links.

In this chapter, we propose a reconfigurable hardware solution, using the well received concept of Binary Decision Diagrams (BDDs), that provides a high-speed IP address lookup and enables a data throughput of 200 Gbps (average packet size of 1000 bits) in the current day large routers [33]. Binary Decision Diagrams (BDDs) are one of the biggest breakthroughs in CAD in the last decade. BDDs are a canonical and efficient way to represent and manipulate Boolean functions and have been successfully used in numerous CAD applications. Although the basic idea has been around for more than 30 years [1], it was Bryant who described a canonical BDD representation [8] and efficient implementation algorithms [6].

The rest of the chapter is organized as follows. Section II presents an overview of the longest prefix matching problem. In Section III, we review the related work done followed by a brief overview on Binary Decision Diagrams and the motivation for the proposed

scheme. Section IV gives the details of the proposed scheme and the implementation issues. In Section V, we present the results obtained from the implementation and analyze the performance of the scheme. We also discusses in detail, the routing table update scheme and the scalability of the scheme to IPv6. Finally, Section VI concludes the discussion.

6.2 Longest Prefix Matching

The routing of the communication packets in the IP domain is done on the Next-Hop basis, i.e., the router takes the responsibility of sending any incoming packet till the next hop only. Consequently, the packet reaches its final destination in multiple hops. The next hop for a packet is determined by the router based on its destination IP address. Each router has a database, in the form of a routing table, of the prefixes of varying length and the corresponding next hop port (NHP) for each prefix. A typical routing table is shown in Table 6.1.

Table 6.1 A sample routing table.

Prefix	*length*	*NHP*
*	0	0
0*	1	1
01*	2	3
10*	2	2
001*	3	1
101*	3	2

The length of the prefixes can vary from 0 to 32 bits. For an incoming packet, its destination address is compared with all the current prefixes in the routing table and the next hop port (NHP) associated with the longest matching prefix is determined to be the output port for the packet. For an example shown in Figure 6.1, a destination IP address 129.186.200.205 matches three prefixes 129/8 (prefix/length), 129.186/16 and 129.186.192/20 in which case, the longest matched prefix 129.186.192/20 is considered

to be the best match and the packet is routed to the output port associated with that particular prefix. In other words, routing based on the longest prefix matching is equivalent to routing the packet to the nearest possible IP address. If none of the prefixes match with the destination IP address, the packet is sent to a default port, which is associated with a prefix of length zero.

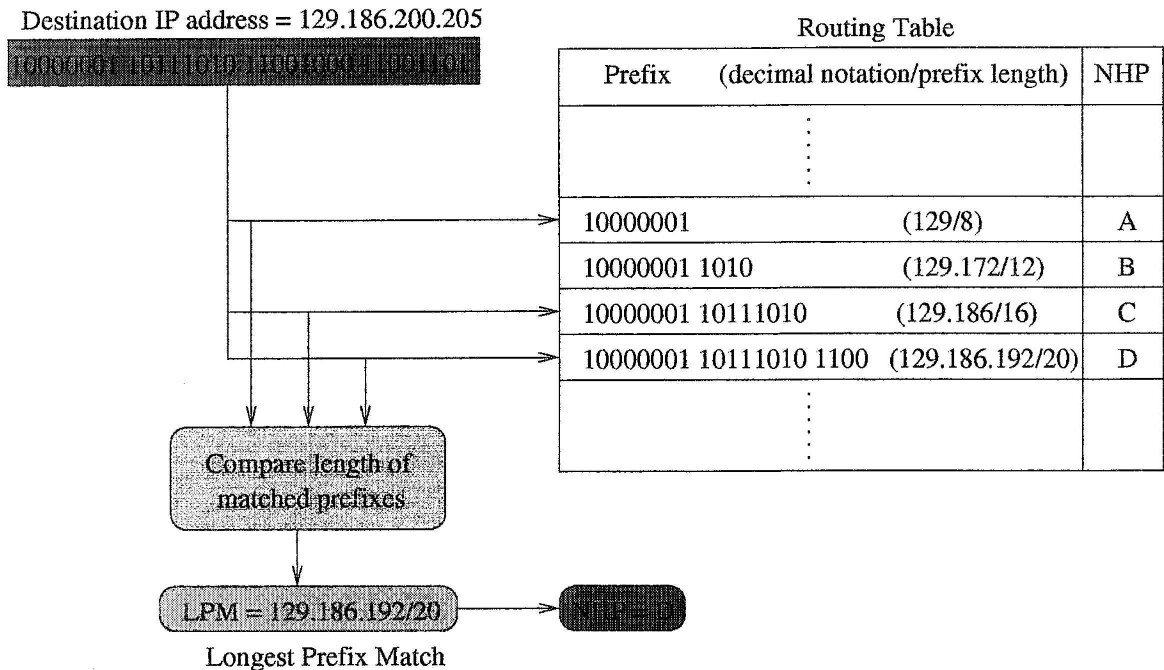


Figure 6.1 Packet routing based on longest prefix matching mechanism.

The metrics taken into consideration, in general, while designing the IP lookup algorithms are *Preprocessing time*, *Storage requirements*, *Lookup rate* and *Update time.* Lookup rate is the most significant parameter that needs to be addressed. With the latest advancements in the network technology, the communication speed is leaping from Ethernet of 10Mbps to Fiber Distributed-Data Interface (FDDI) of 100Mbps to gigabit Ethernet. With the OC192c Line (Line-rate 10Gbps), 31.25 million packets (average size of 40Bytes) have to be processed each second, while for the OC768c (Line-rate 40Gbps), the processing rate required is 125 million packets per second. The data throughput rates of various transmission links and the corresponding time budget for packet processing in a network processor is shown in Table 6.2. It is significant to note that, apart

from the lookup and forwarding operation the packet processing in a network processor includes various other functions like *Protocol recognition and classification*, *Segmentation assembly and reassembly* (SAR), *Queuing and access control*, and *Quality of service* (QoS). The time budget shown for packet processing includes the execution of all these functions, and the lookup operation is required to consume a portion of that budget. Hence, the significance of designing a mechanism for high-speed IP address lookups cannot be overemphasized.

Table 6.2 Data throughput and packet processing time budgets for ATM over SONET. Packet size considered is 40 bytes.

Media	*link rate*	*packets/sec (Million)*	*time per packet (ns)*
OC3	~ 150 Mbps	~ 0.491	~ 2034.5
OC12	625 Mbps	~ 2	~ 488.2
OC48	2.5 Gbps	~ 8.38	~ 119.2
OC192	10 Gbps	~ 33.5	~ 29.8
OC768	40 Gbps	~ 134.2	~ 7.45

The large IPv4 routing tables known today typically contain around 50,000 prefixes and a large IPv6 routing table is expected to contain around 500,000 prefixes. Consequently, the need for enormous amount of data processing at phenomenal speeds, based on longest prefix matching in a large database of prefixes, makes the problem more complicated. Further, the IP address, which is 32 bits long in IPv4, would be 128 bits long, when IPv6 is introduced, making the problem of IP forwarding even more complex.

6.3 Related Research

The IP address lookup schemes introduced so far can be broadly classified into two categories, viz., *software* and *hardware* approaches [61]. In amelioration to the classical binary trie traversal approach, several software solutions have been proposed. One of the first was the prefix matching algorithm using *path-compressed tries* [64] based on the PATRICIA (Practical Algorithm to Retrieve Information Coded in Alphanumeric) trie

introduced in 1968 [51]. The other schemes subsequently proposed include the various trie based approaches [40, 74, 84] and other binary search methods like *binary search on prefix lengths* [77, 76] and *multiway and multicolumn search* [46]. Besides, other schemes based on *prefix expansion* [66], *string matching* [21] and *level compression tries* [55] have been proposed. The software address lookup schemes mostly are based on the tree traversal approach and hence perform according to the computing environment used for the implementation of the algorithm. The key elements that play a pivotal role in the performance of the software mechanism are the processor speed and the memory characteristics (capacity and access time) of the computing environment in which the algorithm is implemented. The best known algorithm for IP address lookups is the *binary search on prefix lengths* with the complexity of the lookup operation being logarithmic in the prefix length (W), i.e., $\mathcal{O}(\log W)$, independent of the number of prefixes (N). Even in such case, the lookup operation involves five memory accesses in the worst case for IPv4 and thus is limited by the capacity of the computing environment. The scheme with $\mathcal{O}(\log W)$ lookup complexity gives a throughput up to 10 Million lookups per second (Mlps), when implemented in a computing environment consisting of a Pentium-Pro-based computer with a clock speed of 200 MHz and a 512 KB L2 cache memory [61]. The throughput, even when scaled for faster processors and larger memory capacity, will not be able to meet the current day requirements in packet processing rates.

Apart from the above software schemes, attempts have been made to design hardware mechanisms for prefix matching to enable high-speed routing. Various hardware schemes like Content Addressable Memories (CAMs) [50], memory lookup based schemes [28, 53, 79], CPU caching [14], and circuit logic implementation in FPGAs [32] have been proposed. Recently, Pao *et al.* proposed a hardware architecture [57] implemented with the partition of binary trie into multiple levels, and Taylor *et al.* proposed a reconfigurable device based Fast Internet Protocol Lookup (FIPL) engine [69] for high-speed routing.

Content Addressable Memory can search all of its entries in parallel given a destination IP address. The scheme based on CAMs uses a separate CAM for each possible

prefix length and hence will require 32 CAMs for IPv4 and 128 CAMs for IPv6 resulting in an expensive solution. Besides, CAM might not be able to keep pace with the fast developing high speed networks as it depends on and is limited by the IC process technology. Memory lookup schemes are based on SRAM indirect indexing, and hence require an additional ASIC to corporate with the memory. The basic scheme in [28] uses a two-level multibit trie with a fixed stride of 24-bits and 8-bits for the first and second level, respectively. This scheme is developed based on the important observation that in a typical backbone router, most of the prefixes are of length 24 bits or less. Thus a prefix expansion methodology is used wherein all the prefixes with length less than 24 bits are expanded accordingly. The memory access speed might not be able to cope up with the advent of new optical link rates and hence limits its performance. The architecture with hardware indexing implementation [57] of binary trie promises a high throughput, but requires a large memory for the implementation. Consequently, the practicality of the implementation of the scheme for the next generation Internet routing with IPv6 protocol remains to be seen. The FIPL engine [69] gives an average throughput of 10 Mlps using Eight FIPL engines for routing in the MAE-West router with 16,564 prefixes. Besides, the scheme does not discuss its scalability to the future trends in Internet routing. In this discussion, we have shown a higher throughput of up to 168.6 Mlps for MAE-West router with a larger number of prefixes (29,487), with an added advantage that our scheme is easily scalable for the rapidly expanding routing tables.

In the past, caching has not worked well in backbone routers because of the need to cache full addresses. This potentially dilutes the cache with hundreds of addresses that map to the same prefix. Besides, typical backbone routers may expect to have hundreds of thousands of flows to different addresses. The Wilder study [70] reports up to 240,000 concurrent flows with less than 20 packets per flow. Short web transfers are a likely reason for this behavior. Some studies have shown cache hit ratios of around 50-70% [54]. Thus, caching can help, but does not avoid the need for fast lookups. Most important, the above schemes become impractical at the advent of IPv6 due to

the requirement of larger storage capacity.

6.3.1 *Binary Decision Diagrams*

As is well-known, a Boolean function f: $B^n \rightarrow$ B can be represented by a Binary Decision Diagram (BDD), a directed acyclic graph obtained by applying an ordering constraint over the input variables and reduction operations on a binary decision tree, as proposed by Bryant [8]. For example, the binary decision tree and the diagram for the function $f = x_0x_1 + x_1x_2 + x_2x_0$ are as shown in Figure 6.2.

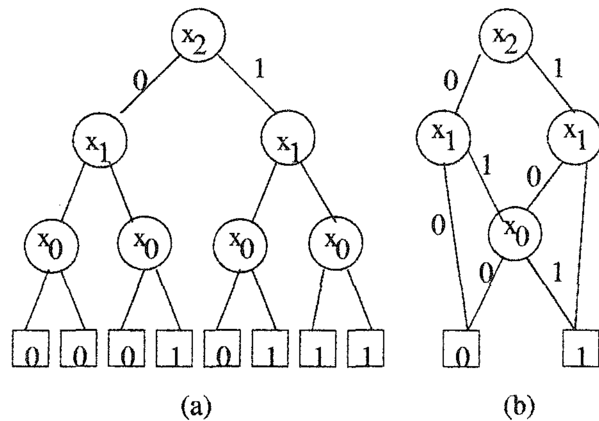


Figure 6.2 Function $f = x_0x_1 + x_1x_2 + x_2x_0$ represented as (a) Binary decision tree and (b) BDD.

Further more, the complexity of the BDD is dependent on its size, measured in the number of nodes. Hence, since a long time, one of the main research focuses has been to reduce the number of nodes in BDD representation. Reduction operations consist in eliminating redundant nodes from the binary tree. A node can be eliminated if

a. both the child nodes are equivalent, which means that the binary logic extracted from both the nodes leads to a same output.

b. there exists an other node at the same level in the decision tree and with equivalent high and low child nodes, respectively.

6.4 BDD Based IP Address Lookups

6.4.1 Motivation

The proposed scheme is motivated from two observations, first being that even at the largest Network Access Point, the number of next hop ports (NHPs) is generally not greater than 256. Hence, a next hop port (NHP) associated with any prefix in the routing table can be encoded using a 8-bit binary code. For example, any next hop port (NHP) in the MAE-east [33] routing table can be safely represented by a 6-bit binary code. Every bit of the output port can be computed by a combinational logic circuit whose optimal minimization is obtained with the help of Binary Decision Diagrams. The second observation is that the number of *effective nodes*, defined as the minimal number of nodes required to construct a binary decision tree in order to cover all the prefixes in the routing table, is significantly smaller as compared to the upper bound on the theoretically required number of nodes. It is shown in the following section that, for the 32-bit IP address with the biggest available routing table of MAE-east [33], the number of redundant nodes is more than 99.99%. Thus constructing the binary decision tree with a fewer nodes and without any redundant nodes makes it very attractive for the application of Binary Decision Diagrams to optimize the logic. Besides, it is shown in the next sections that, while the upper bound on nodes increases exponentially with the IP address size, the number of *effective nodes* do not, making it an advantageous fact in view of the future implementation of IPv6 with the 128-bit IP address.

6.4.2 Details of the Scheme

For further understanding, consider the routing table given in Table 6.1. The binary decision tree representation for the routing table is shown in Figure 6.3. A node is assigned with the associated next hop port (NHP) if the path taken till that node from the root node forms a valid prefix. Note that the root node is assigned with a default output port (in this case 0) as the length of the prefix at that node is zero. However, as mentioned earlier, the partial construction of the binary decision tree is sufficient to cover

all the prefixes in the routing table resulting in only eight *effective nodes.* The redundant nodes, which can be conveniently ignored in the binary decision tree representation, are shown in dotted lines.

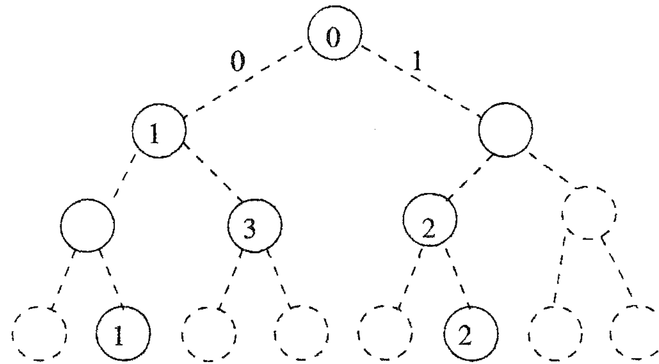


Figure 6.3 Binary decision tree for the sample routing table. Dotted nodes are redundant.

Now, the number of distinct next hop ports in this case being four, each next hop port (NHP) is encoded with a 2-bit binary code, NHP_1 and NHP_0 being the most significant (MSB) and the least significant (LSB) bits, respectively. When the ports are identified with the binary code, the binary decision tree representations for the NHP_0 and NHP_1 bits are as shown in Figure 6.4. It can be observed that a further reduction in the number of *effective nodes* is obtained, the process of which is explained in detail in the subsequent subsection. Note that any effective node without an output bit assigned to it, would inherit the output of its parent node.

For the sake of convenience, let's assume that the n-bit IP address is represented by the binary variables x_{n-1}, x_{n-2}, ..., x_0 where x_{n-1} represents the MSB of the IP address. Now, applying the BDD algorithm on the binary decision trees of each bit of the output port, the BDDs for the functions are obtained to be as shown in Figure 6.5.

6.4.3 *Reducing Effective Nodes*

When the output port is assigned to each of the node on the binary decision tree, with the further analysis, it is observed that the binary encoding of the output ports has

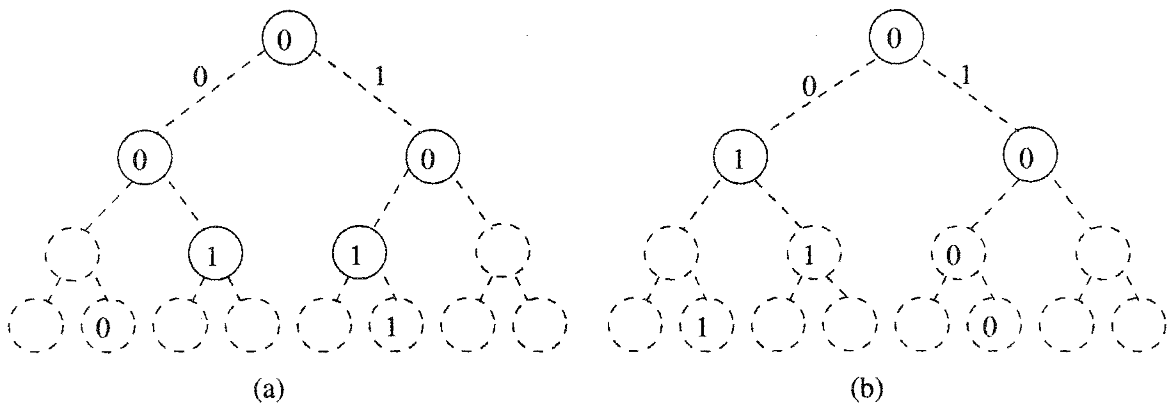


Figure 6.4 Binary decision tree for (a) NHP_1 (b) NHP_0, with all effective nodes assigned with output. Dotted nodes are redundant.

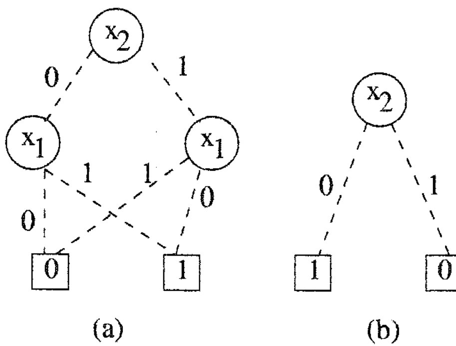


Figure 6.5 BDDs for (a)NHP_1. (b)NHP_0

given a further scope for the reduction in the number of nodes. For example, consider a situation where two leaf nodes, with a common parent, are assigned with output ports of 3 and 11, respectively. Suppose the parent node is assigned with an output port of 2. When the next hop port is encoded with a 4-bit binary code, it can be observed, as shown in Figure 6.6, that a child node with the same output bit as its parent becomes redundant. The redundant nodes are shown in dotted lines in the figure.

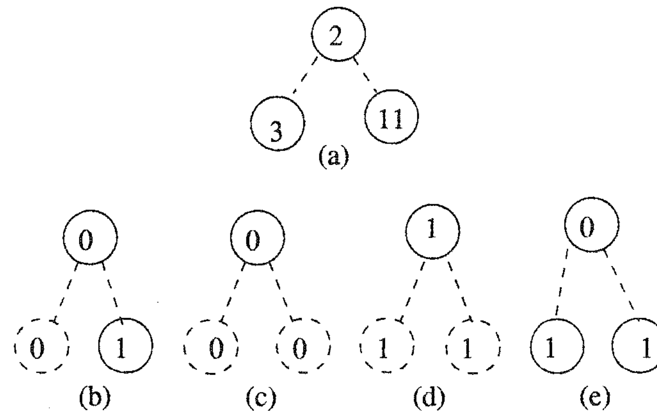


Figure 6.6 (a) Nodes with assigned NHP ports. (b),(c),(d),(e) Output bits assigned to each of the nodes in 4-bit binary encoding of NHP. Dotted nodes are redundant.

With the above procedure, it is shown that in each of the output bit representation, for the biggest available routing table of MAE-east [33] with 32-bit IP address, an additional 36% reduction is obtained in the number of effective nodes. This significant reduction in the number of effective nodes makes the application of Binary Decision Diagram approach, for obtaining the optimized logic, even more effective.

The number of effective nodes are obtained during the construction of the binary decision tree for a few sample routing tables with IP address lengths of 3, 5, 8 and 16 and for the real-time 32-bit IP MAE-east routing table. The prefix distribution of a real-time routing table available at [33], has enabled to generate the prefixes in similar lines for the sample routing tables with IP address lengths of 3, 5, 8 and 16. It is observed that the construction of the binary decision tree for the MAE-east routing table required only 91925 effective nodes, which is largely insignificant as compared to the theoretical

upper bound of more than eight billion nodes. Further, when the output port is encoded with a 6-bit binary code and the reduction procedure is applied on each of the trees for individual binary output bits, the number of effective nodes obtained were only around 64% of the actual effective nodes. The summary of results is shown in Table 6.3.

Table 6.3 Effective nodes for sample routing tables and the real-time 32-bit IP MAE-east routing table with 24792 prefixes.

IP address length	Effective nodes before reduction	Effective nodes after reduction					
		NHP_5	NHP_4	NHP_3	NHP_2	NHP_1	NHP_0
3	8	-	-	-	-	5	3
5	31	-	-	-	20	18	20
8	116	-	-	85	87	81	85
16	3408	-	2098	2172	2181	2219	2164
32 (MAE-east)	91925	57955	58781	58577	58409	58387	58712

6.4.4 Implementation Issues

As mentioned earlier, the output interface ports at any router can be identified by at most an 8-bit binary code. Hence, for the above proposed scheme, the combinational logic design has to be done for eight output bits and hence that would give eight Binary Decision Diagrams to be processed. Each of the synthesized logic can be mapped into one or more Configurable Logic Blocks (CLBs) in an FPGA as shown in Figure 6.7. The processing of the BDDs is performed with the SIS package [23]. The combinational logic subsequently obtained is implemented using Verilog coding and the logic synthesis is performed using the design analyzer tools from Synopsys [35].

6.4.4.1 Timing Optimization

Since the logic design obtained for the IP routing table is a combinational circuit, the timing optimization can be achieved using the *pipelining* and *retiming* techniques. Pipelining involves the insertion of delay elements at specific points of a circuit and

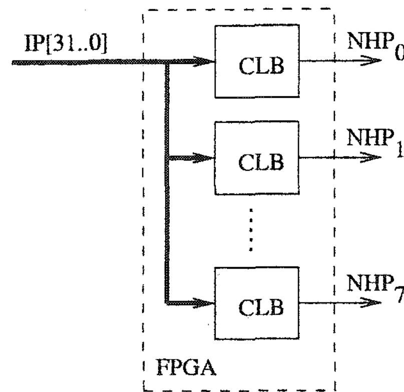


Figure 6.7 CLB mapping in FPGA.

retiming is the process of moving delays around a circuit such that the overall computation is unaltered. It aims to move a computation in an attempt to reduce the critical path, the path with the longest computation time without delays. By pipelining the computational data path, the throughput in terms of number of address lookups per unit time can be increased with a little or no additional cost in the overall area and latency.

6.5 Results and Analysis

The recent research in logic optimization [22, 3] using BDDs has proved that the logic implementation, with a binary decision tree size of more than 100,000 nodes is done in less than a second. Subsequently, it is an encouraging factor when the routing table, with only around 50,000 effective nodes on average, is implemented as a combinational logic optimized using BDDs. While the complexity metrics are important for assessing the feasibility of the implementation, it is equally important to measure the performance of the schemes for real-time routing tables. We measured the performance of our scheme with prefix database of real-time snapshots of various routing tables [33]. The implementation of the routing mechanism is performed as discussed earlier. The lookup time is measured as the propagation delay between the input and output ports of the combinational logic. This is same as the time taken for signal propagation along

the critical path between the input and outports of the logic. The critical path exists between one of 32 input signals and one of eight outputs. Thus, this measurement of propagation delay gives the worst-case lookup time. The worst-case lookup time and the corresponding packet throughput for the proposed scheme, for different routers, are shown in Table 6.4.

Table 6.4 Lookup time performance analysis of BDD based routing engine. Throughput is number of packets per second.

Router	*Prefix count*	*lookup time (ns)*	*throughput (Million)*
MAE-west	29487	5.93	168.63
MAE-east	24792	5.81	172.11
AADS	33796	5.69	175.74
PacBell	6822	4.36	229.35

A main advantage with the proposed scheme is that, for an n-bit binary encoding 2^n number of output ports can be represented and hence, with an increase by one bit in the binary code twice the current number of output ports can be represented. Thus, the proposed scheme proves to be more beneficial in the scenario that the number of physical ports in a router would increase continuously. Besides, when the routing table is implemented in an FPGA, we can conclude that the IP address lookup rate is bounded only by the CLB delay. The maximum clock period bound for processing the IP address lookup would be the sum of 1 CLB delay and the maximum net delay. Previous hardware schemes have the lookup rate bounded by the RAM access speed. Further, in this scheme the resources required are utmost one FPGA while the other schemes require an ASIC and 3 or 4-bank RAM.

6.5.1 Routing Table Update

As discussed in the earlier sections, the routing table update time is one of the important metrics to be considered for a scheme attempting the IP address lookup problem. In earlier hardware schemes for IP address lookups, the update scheme is

based on the assumption of availability of redundant hardware resources, which can be a duplicate memory bank [28] or Content Addressable Memories (CAMs). When one unit is actively involved in the routing of packets, the redundant unit is used by the backbone router to update the routing table offline. The two units are switched alternatively for routing mechanism in a periodical fashion. In our scheme too we assume a similar mechanism. In this scheme, we show that when there is an update in the routing table, then in most cases, not all of the logic blocks have to be recomputed, thus reducing the computational complexity. For example if a prefix 11* is inserted with an associated next hop port to be 1 into the routing table shown in Table 6.1, then the new routing table would be as shown in Table 6.5.

Table 6.5 Modified routing table.

Prefix	*length*	*NHP*
*	0	0
0*	1	1
01*	2	3
10*	2	2
11*	2	1
001*	3	1
101*	3	2

The binary decision tree for the modified routing table would be as shown in Figure 6.8(a). It is obvious that there is no change in the BDD representation for the output bit NHP_1 while the slightly modified BDD representation for output bit NHP_0 is as shown in Figure 6.8(b).

However, this update of the logic may not be that simple for the 32-bit IP MAE-east routing table, but is also not as complex as assumed in general. To demonstrate this simplicity in updating the routing table, we have considered two consecutive snapshots of the MAE-east routing tables from [33], with the number of prefixes 19477 and 19525, respectively. The analysis for the updating of the table is done in terms of the number of nodes at each level, in the binary decision tree for the latter routing table that differ

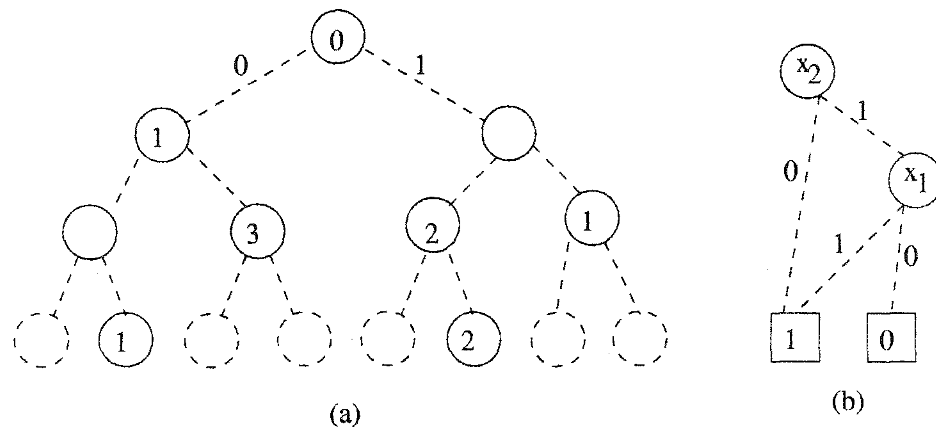


Figure 6.8 (a) Binary decision tree representation of the modified routing table. Dotted nodes are redundant. (b) Modified BDD for NHP_0.

in the output as compared to the corresponding nodes in the binary decision tree for the former routing table. Encouraging results have been obtained during this analysis and the results are shown in Table 6.6.

It is interesting to note that there is none or a significantly smaller variation in the output between the consecutive routing tables at the higher levels (level 0 to 15) and the lower levels (level 25 to 31). As is commonly known, the change in the logic would be minimal when the changes are minimum at higher levels in the binary decision tree, and we can observe that the same is the case in the current scenario. Further more, it can be observed that the number of nodes, in the levels 16 to 24, that differ in their outputs, are significantly smaller. Based on the observations, it can be safely concluded that, when the routing table is updated, there would only be a partial change in the combinational logic for each of the output bit. Thus the reconfiguration of only those logic segments, that need to have the updated logic, can be done. The commercial availability of partially reconfigurable FPGAs makes this update scheme even more attractive, where in, only those CLBs that have a modified design can be reconfigured leaving the remaining CLBs unaltered.

Table 6.6 Number of corresponding nodes in each level of binary decision trees that differ in their output. The two binary decision trees compared are for adjacent snapshots of real-time MAE-east routing table.

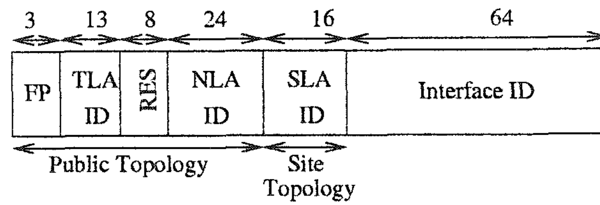
Level	Number of nodes					
	NHP_5	NHP_4	NHP_3	NHP_2	NHP_1	NHP_0
0	0	0	0	0	0	0
1	0	0	0	0	0	0
2	0	0	0	0	0	0
3	0	0	0	0	0	0
4	0	0	0	0	0	0
5	0	0	0	0	0	0
6	0	0	0	0	0	0
7	0	0	0	0	0	0
8	1	1	0	2	2	0
9	0	0	2	1	1	1
10	0	0	0	1	1	0
11	0	0	0	1	1	0
12	0	1	1	3	3	1
13	0	2	2	4	4	2
14	0	4	4	6	6	4
15	0	8	7	8	8	8
16	1	16	14	14	15	17
17	2	24	23	25	23	24
18	6	35	32	33	33	37
19	20	36	37	40	38	35
20	9	8	10	6	11	5
21	5	4	7	5	6	3
22	7	3	5	4	4	6
23	8	11	10	10	11	7
24	31	42	34	34	39	31
25	0	0	0	0	0	0
26	0	0	0	0	0	0
27	0	0	0	0	0	0
28	0	0	0	0	0	0
29	0	0	0	0	0	0
30	0	0	0	0	0	0
31	0	0	0	0	0	0

6.5.2 Scalability to IPv6

To demonstrate that an optimized combinational logic can be obtained for a mapping between 128-bit long IP address of IPv6 and the binary encoded next hop port, it would be appropriate to show that the number of effective nodes that need to be processed by the BDD reduction techniques is significantly smaller than the theoretical upper bound. The performance analysis of our scheme in IPv4 was feasible since real-time 32-bit routing table were readily available [33]. However, a similar analysis was not possible in IPv6 due to the nonavailability of the routing table and hence, to start with, we had to construct a routing table in line with the specifications of IPv6 protocol.

The building of the IPv6 routing table is done by incorporating the best-effort unicast address called *aggregatable global unicast address* [5]. This address format was designed to facilitate scalable Internet routing, by providing an address hierarchy flow aggregation. The address format has a fixed structure as shown in Figure 6.9 and is organized into a three level hierarchy: Public Topology; Site Topology; and Interface Identifier. The Public Topology consists of a two level hierarchy of service providers with a Top-Level Aggregation Identifier (TLA ID) and a Next-Level Aggregation Identifier (NLA ID). The TLA ID is initially to be restricted to 13 bits which translates to 8192 routers in the core IPv6 network. This was done to constrain core routing table sizes. The NLA ID is 24 bits long and allows for a flat or hierarchical allocation of the NLA address space. The Site-Level Aggregation Identifier (SLA ID) is 16 bits long. It is used by an individual organization to define its local address hierarchy and subnets.

The routing table constructed using the described IPv6 address format constituted a large database of 400,000 prefixes with prefix length ranging from 0 to 128 bits. The number of output ports were 512 and hence a 9-bit binary code is used to encode the NHP. The database is built such that the prefix length distribution in the IPv6 routing table should ratify the hierarchical topology of aggregatable global unicast address. The number of effective nodes are obtained during the construction of the binary decision tree for the IPv6 routing table with 128-bit long address. It is observed that the construction of the binary decision tree for the IPv6 routing table required only 13.5×10^6 effective



FP: Format Prefix , TLA ID: Top–Level Aggregation Identifier
RES: Reserved for future use, NLA ID: Next–Level Aggregation ID
SLA ID: Site–Level Aggregation ID

Figure 6.9 Aggregatable Global Unicast Address for IPv6.

which is enormously insignificant as compared to the theoretical upper bound of 6.8 x 10^{38}. Further, when the output port is encoded with the binary code and the reduction procedure is applied on each of the trees for individual binary output bits, the number of effective nodes obtained were only 7 x 10^{6}, around 51% of the actual effective nodes. Thus the significantly smaller number of nodes that need to be processed in the BDD solving of the logic shows that the scheme is scalable for the forthcoming IPv6. The implementation of the routing table and the measurement of the lookup time in IPv6 routing is reserved for our future study.

6.6 Summary

With the advancements in the communication link technologies the IP address lookup is becoming a major bottleneck in router technologies. We propose a reconfigurable hardware solution, using the well received concept of Binary Decision Diagrams(BDDs), that provides an efficient IP address lookup along with providing a better scheme for updating the routing table. The argument, to support the adoption of BDD techniques for obtaining an optimized combinational logic, has been put forward by showing the fact that the number of effective nodes required to represent a 32-bit IP address routing table is significantly smaller than the theoretically required number of nodes. Besides, it has been shown that this number of effective nodes can be further reduced when the

next hop port is represented with a binary code and a tree representation is obtained for each of the output bits. The implementation of the routing scheme shows that the BDD hardware engine gives a throughput of up to 172.1 Million lookups per second (Mlps) for a large MAE-east routing table with 24,792 prefixes, a throughput of up to 168.6 Mlps for an MAE-west routing table with 29,487 prefixes, and a throughput of up to 229.3 Mlps for the Pacbell routing table with 6,822 prefixes. Thus, a data throughput of 200 Gbps (with an average packet size of 1000 bits) can be obtained in the router implemented with the BDD based hardware address lookup engine.

Following the implementation of the scheme with the analysis of its performance in terms of the lookup time and packet throughput in IPv4 routing, and the proof that the processing time for the logic optimization in IPv6 routing tables is well under limit due to the emphatically smaller number of effective nodes, the next step is to obtain an implementation of the scheme for IPv6 and measure the lookup time.

7 CONCLUSIONS

Applications that demand either higher on-chip computing power or larger on-chip memory bandwidth are continuously emerging. Processor researchers and designers are continuously facing the uphill task of simultaneously meeting these two demands by various applications. In this dissertation, we proposed *Adaptive Register file* architecture, a novel architecture to provide a feasible solution and address the above problems concurrently. In the first phase, *TriBank Register file*, an effective register file organization that aims to simultaneously meet two main goals of providing a small register access time to enable a faster processor cycle time, and provide a large number of registers to enable dispatching as many instructions as possible to issue window for extracting higher ILP. Implementation of the TriBank register file organization, as compared to a conventional monolithic register file in an 8-wide out-of-order issue superscalar processor enhanced the throughput in instructions per cycle (IPC) by 3% and 14%, for SpecInt2000 and SpecFP2000, respectively. When the register file access time is factored in, the instruction throughput is enhanced up to 56% and 96%, for SpecInt2000 and SpecFP2000, respectively.

In the second phase, the *Adaptive Register file Computing* (ARC) unit is developed, to provide a higher on-chip computing capacity by executing a compute-intensive function, and to provide a larger register file resources to meet the memory bandwidth requirements. Results show a performance increase of up to 12%, when an out-of-order 8-wide issue superscalar processor is supplemented with the ARC unit to process matrix multiplication, a compute-intensive core function in most multimedia applications. The dissertation also discussed the microarchitecture level details for the implementation of the ARC unit.

The dissertation further discussed the high performance of the Reconfigurable Functional Cache (RFC) based processor in computing various multimedia applications, and its ability to deliver such high performance even while consuming lesser amount energy. Further, in continuance of our effort to build efficient adaptive architectures with improved performance, various RFC configuration schemes have been studied. A detailed study of the performance analysis of the architecture, in terms of the execution time of the application, is given. The impact of various architectural parameters and the factors governing the structure of an application over the execution time of an application has been extensively studied. With the help of the study undertaken, the design of ABC microprocessor can be incorporated with the dynamic decision capability, so that appropriate RFC configuration scheme is chosen dynamically for running a particular application.

The dissertation also discussed a compute intensive application that forms a bottleneck in the design of an effective network processor. The dissertation proposed a reconfigurable hardware solution, using the well received concept of Binary Decision Diagrams(BDDs), that provides an efficient IP address lookup along with providing a better scheme for updating the routing table. The implementation of the routing scheme shows that the BDD hardware engine gives a throughput of up to 172.1 Million lookups per second (Mlps) for a large MAE-east routing table with 24,792 prefixes, a throughput of up to 168.6 Mlps for an MAE-west routing table with 29,487 prefixes, and a throughput of up to 229.3 Mlps for the Pacbell routing table with 6,822 prefixes. Thus, a data throughput of 200 Gbps (with an average packet size of 1000 bits) can be obtained in the router implemented with the BDD based hardware address lookup engine.

7.1 Future Research

Further investigations involve the study of performance of proposed *Adaptive Register file* architecture in executing various multimedia and signal processing applications that have the matrix multiplication as the core compute-intensive function. The study also requires the development of a compiler that automatically extracts the matrix mul-

tiplication function in an application and generates a suitable sequence of instructions to perform the computation in the ARC unit. A mechanism, with a suitable support at the compiler and the microarchitecture level, needs to be devised to support the dynamic allocation of the ARC resources for computation and register file purposes. With the help of such mechanism, an effort can be made towards achieving a balanced computation for higher performance in superscalar processors.

Following the implementation of the IP address lookup scheme, with the analysis of its performance in terms of the lookup time and packet throughput in IPv4 routing, and the proof that the processing time for the logic optimization in IPv6 routing tables is well under limit due to the emphatically smaller number of effective nodes, the next step is to obtain an implementation of the scheme for IPv6 and measure the lookup time.

BIBLIOGRAPHY

[1] S. B. Akers. Binary decision diagrams. *IEEE Transactions on Computers*, Vol. c-27, No. 6:509–516, June 1978.

[2] R. Balasubramonian, S. Dwarkadas, and D. H. Albonesi. Reducing the complexity of the register file in dynamic superscalar processors. *Proc. 34th ACM/IEEE International Symposium on Microarchitecture, MICRO-34*, pages 237–248, 2001.

[3] V. Bertacco and M. Damiani. The disjunctive decomposition of logic functions. *Proc. IEEE/ACM International Conference on Computer-Aided Design, ICCAD*, pages 78–82, 1997.

[4] E. Borch, E. Tune, S. Manne, and J. Emer. Loose loops sink chips. *Proc. Eighth International Symposium on High-Performance Computer Architecture*, pages 270–281, 2002.

[5] P. Boustead and J. Chicharo. Label switching using the ipv6 address hierarchy. *Proc. IEEE Global Telecommunications Conference, GLOBECOM 2000*, Vol. 1:500–504, 2000.

[6] K. S. Brace, R. L. Rudell, and R. E. Bryant. Efficient implementation of a bdd package. *Proc. 27th IEEE/ACM Design Automation Conference*, pages 40–45, 1990.

[7] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A framework for architectural-level power analysis and optimizations. *Proc. 27th International Symposium on Computer Architecture*, pages 83–84, 2000.

[8] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, Vol. c-35:677–691, August 1986.

[9] Doug Burger and Todd M. Austin. The simplescalar tool set, version 2.0. *Computer Sciences Department Technical report*, 1342, 1997.

[10] Doug Burger, James R. Goodman, and Alain Kgi. Memory bandwidth limitations of future microprocessors. *Proc. 23rd Annual International Symposium on Computer Architecture*, pages 78–89, May 1996.

[11] T. J. Callahan, J. R. Hauser, and J. Wawrzynek. The garp architecture and c compiler. *IEEE Computer*, Vol. 33:62–69, April 2000.

[12] Meng-Chou Chang and Feipei Lai. Efficient exploitation of instruction-level parallelism for superscalar processors by the conjugate register file scheme. *IEEE Transactions on Computers*, Vol. 45:278–293, March 1996.

[13] Chung-Ho Chen and A. K. Somani. Architecture technique trade-offs using mean memory delay time. *IEEE Transactions on Computers*, Volume: 45, Issue: 10:1089–1100, October 1996.

[14] T. Chiueh and P. Pradhan. High-performance ip routing table lookup using cpu caching. *Proc. IEEE INFOCOM'99*, Vol. 3:1421–1428, 1999.

[15] J. L. Cruz, A. Gonzalez, M. Valero, and N. P. Topham. Multiple-banked register file architectures. *Proc. 27th Annual International Symposium on Computer Architecture*, pages 316–325, 2000.

[16] W. J. Dally. Interconnect-limited vlsi architecture. *Proc. IEEE International Conference on Interconnect Technology*, pages 15–17, 1999.

[17] A. DeHon. Dpga-coupled microprocessors: commodity ics for the early 21st century. *Proc. IEEE Workshop on FPGAs for Custom Computing Machines*, pages 31–39, 1994.

[18] A. DeHon. Dpga-coupled microprocessors: commodity ics for the early 21st century. *IEEE Workshop on FPGAs for Custom Computing Machines*, pages 31–39, 1994.

[19] A. DeHon. The density advantage of configurable computing. *IEEE Computer*, 33:41–49, April 2000.

[20] A. DeHon and J. Wawrzynek. Reconfigurable computing: what, why, and implications for design automation. *Proc. 36th ACM/IEEE Conference on Design Automation*, pages 610–615, 1999.

[21] A. Donnelly and T. Deegan. Ip route lookups as string matching. *Proc. 25th Annual IEEE Conference on Local Computer Networks, LCN*, pages 589–595, 2000.

[22] R. Drechsler and W. Gunther. Optimization of sequential verification by history-based dynamic minimization of bdds. *Proc. IEEE International Symposium on Circuits and Systems, ISCAS*, Vol. 4:737–740, 2000.

[23] E. M. Sentovich et al. Sis: A system for sequential circuit synthesis. *Memorandum No. UCB/ERL M92/41, Electronics Research Laboratory, University of California, Berkeley*, 2001.

[24] K. I. Farkas, P. Chow, N. P. Jouppi, and Z. Vranesic. Memory-system design considerations for dynamically-scheduled processors. *Proc. 24th Annual International Symposium on Computer Architecture*, pages 133–143, 1997.

[25] K. I. Farkas, N. P. Jouppi, and P. Chow. Register file design considerations in dynamically scheduled processors. *Proc. Second International Symposium on High-Performance Computer Architecture*, pages 40–51, 1996.

[26] G. H. Golub and C. F. Van Loan. Matrix computations. *Johns Hopkins University Press*, USA, 1989.

[27] A. Gonzalez, M. Valero, J. Gonzalez, and T. Monreal. Virtual registers. *IEEE Fourth International Conference on High-Performance Computing*, pages 364–369, 1997.

[28] P. Gupta, S. Lin, and N. McKeown. Routing lookups in hardware at memory access speeds. *Proc. IEEE INFOCOM'98*, Vol. 3:1240–1247, 1998.

[29] J. R. Hauser and T. J. Callahan. Personal communication. University of California - Berkeley, November 2001.

[30] J. R. Hauser and J. Wawrzynek. Garp: a mips processor with a reconfigurable coprocessor. *Proc. IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 12–21, April 1997.

[31] John L. Hennessy and David A. Patterson. Computer architecture: A quantitative approach. *Morgan Kaufman*, California, USA, 1996.

[32] Ilion Yi-Liang Hsiao and Chein-Wei Jen. A new hardware design and fpga implementation for internet routing towards ip over wdm and terabit routers. *Proc. IEEE International Symposium on Circuits and Systems, ISCAS 2000*, Vol. 1:387–390, 2000.

[33] http://www.merit.edu/ipma. Routing-analysis, internet performance measurement and analysis (ipma) project. *online*, February 15, 2001.

[34] M. D. Powell I. Park and T. N. Vijayakumar. Reducing register ports for higher speed and lower energy. *Proc. 35th Annual International Symposium on Microarchitecture*, 2002.

[35] Synopsys Inc. http://www.synopsys.com/. February 10, 2002.

[36] Texas Instruments. Tms320c6000 benchmarks. *http://www.ti.com/sc/docs/products/dsp/c6000/62bench.htm*, October 10, 2000.

[37] R. Joseph and M. Martonosi. Run-time power estimation in high performance microprocessors. *Proc. International Symposium on Low Power Electronics and Design*, pages 135–140, 2001.

[38] W. Kautz. Cellular logic-in-memory arrays. *IEEE Transactions on Computers*, Vol. C-18:719–727, August 1969.

[39] R. E. Kessler. The alpha 21264 microprocessor. *IEEE Micro*, Vol. 19:24–36, March-April 1999.

[40] T. Kijkanjanarat and H. J. Chao. Fast ip lookups using a two-trie data structure. *Proc. Global Telecommunications Conference, GLOBECOM*, Vol. 2:1570–1575, 1999.

[41] H. Kim, A. K. Somani, and A. Tyagi. A reconfigurable multifunction computing cache architecture. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, Vol. 9:509–523, August 2001.

[42] Huesung Kim. Towards adaptive balanced computing (abc) using reconfigurable functional caches (rfcs). *Ph. D. Dissertation, Dept. of Electrical and Computer Engineering, Iowa State University*, July 2001.

[43] A. Kumar. The hp pa-8000 risc cpu. *IEEE Micro*, Vol. 17:27–32, March-April 1997.

[44] H. T. Kung. Memory requirements for balanced computer architectures. *Proc. 13th Annual International Symposium on Computer Architecture*, pages 49–54, 1986.

[45] S. Y. Kung. Vlsi array processors. *Prentice Hall*, USA, 1988.

[46] B. Lampson, V. Srinivasan, and G. Varghese. Ip lookups using multiway and multicolumn search. *Proc. IEEE INFOCOM'98*, pages 1248–1256, 1998.

[47] A. R. Lebeck, J. Koppanalil, Tong Li, J. Patwardhan, and E. Rotenberg. A large, fast instruction window for tolerating cache misses. *Proc. 29th Annual International Symposium on Computer Architecture*, pages 59–70, 2002.

[48] Chunho Lee, M. Potkonjak, and W. H. Mangione-Smith. Mediabench: a tool for evaluating and synthesizing multimedia and communications systems. *Proc. Thirtieth Annual IEEE/ACM International Symposium on Microarchitecture*, pages 330–335, 1997.

[49] J. Llosa, M. Valero, and E. Ayguade. Non-consistent dual register files to reduce register pressure. *Proc. First IEEE Symposium on High-Performance Computer Architecture*, pages 22–31, 1995.

[50] A. McAuley, P. Tsuchiya, and D. Wilson. Fast multilevel hierarchical routing table using content-addressable memory. *U.S. Patent serial number 034444*, 1995.

[51] D. Morrison. Patricia – practical algorithm to retrieve information coded in alphanumeric. *Journal ofthe ACM*, Vol. 15, No. 4:514–534, October 1968.

[52] M. Moudgill, K. Pingali, and S. Vassiliadis. Register renaming and dynamic speculation: An alternative approach. *Proc. 26th Annual International Symposium on Microarchitecture*, pages 202–213, 1993.

[53] J. Pan N. Huang, S. Zhao and C. Su. A fast ip routing lookup scheme for gigabit switch routers. *Proc. IEEE INFOCOM'99*, Vol. 3:1429–1436, 1999.

[54] P. Newman, G. Minshall, and L. Huston. Ip switching and gigabit routers. *IEEE Communications Magazine*, Vol. 35, No. 1:64–69, January 1997.

[55] S. Nilsson and G. Karlsson. Ip address lookup using lc-tries. *IEEE Journal on Selected Areas in Communications*, Vol. 17, No. 6:1083–1092, June 1999.

[56] S. Palacharla, N. P. Jouppi, and J. E. Smith. Complexity-effective superscalar processors. *Proc. 24th Annual International Symposium on Computer Architecture*, pages 206–218, 1997.

[57] D. Pao, C. Liu, A. Wu, L. Yeung, and K. S. Chan. Efficient hardware architecture for fast ip address lookup. *Proc. IEEE INFOCOM'2002*, Vol. 3, 2002.

[58] Keshab K. Parhi. Vlsi digital signal processing systems design and implementation. *Wiley*, New Jersey, USA, 1999.

[59] R. Razdan and M. D. Smith. A high-performance microarchitecture with hardware-programmable functional units. *Proc. 27th Annual International Symposium on Microarchitecture, MICRO-27*, pages 172–180, 1994.

[60] S. Rixner, W. J. Dally, B. Khailany, P. Mattson, U. J. Kapasi, and J. D. Owens. Register organization for media processing. *Proc. Sixth International Symposium on High-Performance Computer Architecture*, pages 375–386, 2000.

[61] M. A. Ruiz-Sanchez, E. W. Biersack, and W. Dabbous. Survey and taxonomy of ip address lookup algorithms. *IEEE Network*, Vol. 15, Issue: 2:8–23, March-April 2001.

[62] R. M. Russell. The cray-1 computer system. *Reading in Computer Architecture, Morgan Kaufman*, pages 40–49, 2000.

[63] P. Shivakumar and N. P. Jouppi. Cacti3.0: An integrated cache timing, power, and area power model. *DEC WRL Research*, 2001/2, 2001.

[64] K. Sklower. A tree-based packet routing table for berkeley unix. *Proc. Winter Usenix Conference*, pages 93–99, 1991.

[65] J. E. Smith and G. S. Sohi. The microarchitecture of superscalar processors. *Proceedings of the IEEE*, Vol. 83:1609–1624, December 1995.

[66] V. Srinivasan and G. Varghese. Fast address lookups using controlled prefix expansion. *Proc. ACM Sigmetrics'98*, pages 1–11, June 1998.

[67] H. S. Stone. A logic-in-memory computer. *IEEE Transactions on Computers*, pages 73–78, January 1970.

[68] J. Swensen and Y. Patt. Hierarchical registers for scientific computers. *Proc. International Conference on Supercomputing*, pages 346–353, 1988.

[69] D. E. Taylor, J. W. Lockwood, T. S. Sproull, J. S. Turner, and D. B. Parlour. Scalable ip lookup for programmable routers. *Proc. IEEE INFOCOM'2002*, Vol. 3, 2002.

[70] K. Thompson, G. J. Miller, and R. Wilder. Wide-area internet traffic patterns and characteristics. *IEEE Network*, Vol. 11, No. 6:10–23, November-December 1997.

[71] M. Tremblay, B. Joy, and K. Shin. A three dimensional register file for superscalar processors. *Proc. 28th Hawaii International Conference on System Sciences*, Vol. II:191–201, 1995.

[72] J. H. Tseng and K. Asanovic. Banked multiported register files for high-frequency superscalar microprocessors. *Proc. 30th Annual International Symposium on Computer Architecture*, 2003.

[73] Dean M. Tullsen, Susan J. Eggers, Joel S. Emer, Henry M. Levy, Jack L. Lo, and Rebecca L. Stamm. Exploiting choice: instruction fetch and issue on an implementable simultaneous multithreading processor. *Proc. 23rd Annual International Symposium on Computer Architecture*, pages 191–202, 1996.

[74] Henry Hong-Yi Tzeng and Tony Przygienda. On fast address-lookup algorithms. *IEEE Journal On Selected Areas In Communications*, Vol. 17, No. 6:1067–1082, June 1999.

[75] Tomohisa Wada, Suresh Rajan, and Steven A. Przybylski. An analytical access time model for on-chip cache memories. *IEEE Journal of Solid-State Circuits*, Vol. 27:1147–1156, August 1992.

[76] M. Waldvogel, G. Varghese, J. Turner, and B. Plattner. Scalable high-speed prefix matching. *ACM Transactions on Computer Systems*, Vol. 19, No. 4:440–482, November 2001.

[77] M. Waldvogel, G. Varghese, J. Turner, and B. Plattner. Scalable high speed ip routing lookups. *Proc. ACM SIGCOMM '97*, Vol. 27, No. 4:25–36, October 1997.

[78] S. Wallace and N. Bagherzadeh. A scalable register file architecture for dynamically scheduled processors. *Proc. Conference on Parallel Architectures and Compilation Techniques*, pages 179–184, 1996.

[79] Pi-Chung Wang, Chia-Tai Chan, and Yaw-Chung Chen. A fast ip routing lookup scheme. *Proc. IEEE International Conference on Communications*, Vol. 2:1140–1144, 2000.

[80] S. E. Wilton and N. P. Jouppi. An enhanced access and cycle time model for on-chip caches. *DEC WRL Research*, 93/5, 1994.

[81] R. D. Wittig and P. Chow. Onechip: an fpga processor with reconfigurable logic. *Proc. IEEE Symposium on FPGAs for Custom Computing Machines*, pages 126–135, 1996.

[82] Mathew Wojko and Hossam ElGindy. Self configuring binary multiplier for lut addressable fpgas. *Proc. Australasian conference on Parallel and Real-Time Systems*, 1998.

[83] Michael E. Wolf and Monica S. Lam. A data locality optimizing algorithm. *Proc. ACM Conference on Programming Language Design and Implementation*, Vol. 26:33–44, May 1991.

[84] N. Yazdani and P. S. Min. Fast and scalable schemes for the ip address lookup problem. *IEEE Conference on High Performance Switching and Routing*, pages 83–92, 2000.

[85] Z. A. Ye, A. Moshovos, S. Hauck, and P. Banerjee. Chimaera: a high-performance architecture with a tightly-coupled reconfigurable functional unit. *Proc. 27th International Symposium on Computer Architecture*, pages 225–235, 2000.

[86] R. Yung and N. C. Wilhelm. Caching processor general registers. *Proc. IEEE International Conference on Computer Design: VLSI in Computers and Processors, ICCD*, pages 307–312, 1995.

[87] V. Zyuban and P. Kogge. The energy complexity of register files. *Proc. International Symposium on Low Power Electronics and Design*, pages 305–310, 1998.

ACKNOWLEDGMENTS

First and foremost, I thank my adviser Prof. Arun K. Somani for his constant guidance, direction, and support. For me, his achievements have been a great source of inspiration, and it has been an enlightening experience to conduct research under his guidance. His ability to judge the work, critique the ideas, and instantaneously put forward the constructive arguments has pushed me to continuously improvise my researching skills.

I am grateful to Dr. Akhilesh Tyagi for many useful discussions and also for serving on my committee. I thank Dr. Chang, Prof. Kothari and Prof. Baca for useful discussions and suggestions, and serving on my committee. I would like to thank Dr. Gyungho Lee for his many insightful comments, and for suggesting useful references. I am always grateful to Dr.Manimaran for his friendly advices and suggestions which has helped me in many ways to bring a shape to my dissertation.

I believe my desire to gain more and more knowledge by pursuing advanced studies was first kindled by the environment provided by my father. His vast collection of books at home has made me take interest in reading and researching the literature in a wide range of subjects. I am always thankful to my father for providing me with such a good and early start.

I feel am lucky to be born as the youngest for one good reason. As both my elder brothers took each step forward in school and finally into the engineering profession before me, the experiences and lessons learned from each step of theirs have made things immensely easy for me. It was an experience I could gain without living through it. I am always thankful to my brothers for providing me with such resources and knowledge.

The wonderful company given to me by my close friends Srinivas (AC), Anil, Ashish, Vadhi and others at REC Warangal, have made me always long for those days again.

Especially, I am always indebted to Srinivas for his constant support in financial, technical, and moral fronts, which has really helped me face various difficulties in the past ten years. Also, I am thankful for the encouragement and support I got from Divakar and my other friends from REC Warangal.

One person who stood by me in my decision to pursue a PhD degree was my sister Latha. I am always grateful to her for the encouragement and support she gave, and for urging me not to have second thoughts about it.

When I met Nagaraj on the first day I arrived in Ames I didn't fully comprehend the impact his company would have on me. The journey with him into the baffling yet enchanting world of ancient *Sanskrit* literature, though I am yet to feel the tip of the iceberg, was truly enthralling until the time he left the place. He was my friend, teacher, and guide, who taught me so many things. The umpteen number of discussions we had on varied subjects like, "meaning of life", literature, music, Upanishads, Advaita and so on, have made me rave for his company. I look forward, and wish for one more opportunity, to journey again in his company before the destination arrives.

I am always indebted to Indu for introducing me to the wonderful world of Carnatic music, during my stay at Iowa State University. She is my good friend and loving sister who always made me happy with the great food she cooks. When the going got tough or when there were a few brief low moments, due to a rejection of paper or for a similar reason, it was she who encouraged me and motivated me to work harder. I have seen her to be the most hard working person working toward a Ph.D degree, and that inspired me to a great extent.

The technical discussions I had with Vadhi, Seongwoo Kim, Huesung Kim, Sriram, Mahadevan and others in the field of Computer architecture have helped my research to a great extent. I am very much thankful to them for their support.

The time I had at the work place was one of quality because of the wonderful company I had in the form of Jing, Pallab, Anirban, Sashi, Murari, Srini, and all the other past and present members of DCNL. Also the company given to me, during my stay at Iowa State University, by Naveen, Swami, Subbu, Sridhar and others was wonderful. I shall

cherish those moments forever.

The support I got from my undergraduate research students Yana, Basem, Seng-Ming, Boon-Siang, Argenis, and Navpreet was immeasurable. Only with their support, I could convert some of my ideas into implementations, and so was able to publish the work. I thank all of them for the excellent job they have done for me.

The absolute assurance and support I got from Neelima, since the day I first met her in October 2002, has helped me gather the last few pieces of my dissertation with a free and happy mind. I am greatly delighted having her by my side, and am always thankful to her for the cheerful and spirited ambiance she provides.

This research was partially supported by the grants from National Science Foundation, ANI9973102 and CCR9900601. They are much appreciated.

BIOGRAPHICAL SKETCH

This dissertation presents a part of the research work done by Rama Sangireddy during the period of August 1999 to July 2003, in pursuit of his Ph.D. degree in Computer Engineering at Iowa State University. Earlier, he had received the B.Tech degree with distinction in Electrical and Electronics Engineering from the Regional Engineering College, Warangal, India, in 1996, and had earned the Master's degree in Electrical Engineering from the University of Missouri-Rolla, USA, in 1999. His research interests include Computer Architecture, Reconfigurable Computing, Computer Communication Networks, and Fault Tolerant Computing. He is currently a member of IEEE.