Technology Impacts of CMOS Scaling on Microprocessor Core Design for Hard-Fault Tolerance in Single-Core Applications and Optimized Throughput in Throughput-Oriented Chip Multiprocessors

by

Fred Allison Bower III

Department of Computer Science
Duke University

Date:______________________
Approved:

____
Professor Daniel J. Sorin, Advisor

____
Dr. Steven W. Hunter

____
Professor Alvin R. Lebeck

____
Professor Christopher L. Dwyer

____
Professor Landon P. Cox

Dissertation submitted in partial fulfillment of
the requirements for the degree of Doctor
of Philosophy in the Department of
Computer Science in the Graduate School
of Duke University

2010

<u>ABSTRACT</u>

Technology Impacts of CMOS Scaling on

Microprocessor Core Design for Hard-Fault Tolerance

in Single-Core Applications and Optimized Throughput in

Throughput-Oriented Chip Multiprocessors

by

Fred Allison Bower III

Department of Computer Science
Duke University

Date:______________________
Approved:

____
Professor Daniel J. Sorin, Advisor

____
Dr. Steven W. Hunter

____
Professor Alvin R. Lebeck

____
Professor Christopher L. Dwyer

____
Professor Landon P. Cox

An abstract of a dissertation submitted in partial
fulfillment of the requirements for the degree
of Doctor of Philosophy in the Department of
Computer Science in the Graduate School
of Duke University

2010

Copyright © 2010
by
Fred Allison Bower III
All rights reserved

Abstract

The continued march of technological progress, epitomized by Moore's Law provides the microarchitect with increasing numbers of transistors to employ as we continue to shrink feature geometries. Physical limitations impose new constraints upon designers in the areas of overall power and localized power density. Techniques to scale threshold and supply voltages to lower values in order to reduce power consumption of the part have also run into physical limitations, exacerbating power and cooling problems in deep sub-micron CMOS process generations. Smaller device geometries are also subject to increased sensitivity to common failure modes as well as manufacturing process variability.

In the face of these added challenges, we observe a shift in the focus of the industry, away from building ever-larger single-core chips, whose focus is on reducing single-threaded latency, toward a design approach that employs multiple cores on a single chip to improve throughput. While the early multicore era utilized the existing single-core designs of the previous generation in small numbers, subsequent generations have introduced cores tailored to multicore use. These cores seek to achieve power-efficient throughput and have led to a new emphasis on throughput-oriented computing, particularly for Internet workloads, where the end-to-end computational task is dominated by long-latency network operations. The ubiquity of these workloads makes a compelling argument for throughput-oriented designs, but does not free the microarchitect fully from latency demands of common workloads in enterprise and desktop application spaces.

We believe that a continued need for both throughput-oriented and latency-sensitive processors will exist in coming generations of technology. We further opine that making effective use of the additional transistors that will be available may require different techniques for latency-sensitive

designs than for throughput-oriented ones, since we may trade latency or throughput for the desired attribute of a core in each of the respective paradigms.

We make three major contributions with this thesis. Our first contribution is a fine-grained fault diagnosis and deconfiguration technique for array structures, such as the ROB, within the microprocessor core. We present and evaluate two variants of this technique. The first variant uses an existing fault detection and correction technique whose scope is the processor core execution pipeline to ensure correct processor operation. The second variant integrates fault detection and correction into the array structure itself to provide a self-contained, fine-grained, fault detection, diagnosis, and repair technique.

In our second contribution, we develop a lightweight, fine-grained fault diagnosis mechanism for the processor core. In this work, we leverage the first contribution's methods to provide deconfiguration of faulty array elements. We additionally extend the scope of that work to include all pipeline circuitry from instruction issue to retirement.

In our third and final contribution, we focus on throughput-oriented core data cache design. In this work, we study the demands of the throughput-oriented core running a representative workload and then propose and evaluate an alternative data cache implementation that more closely matches the demands of the core. We then show that a better-matched cache design can be exploited to provide improved throughput under a fixed power budget.

Our results show that typical latency-sensitive cores have sufficient redundancy to make fine-grained hard-fault tolerance an affordable alternative for hardening complex designs. Our designs suffer little or no performance loss when no faults are present and retain nearly the same performance characteristics in the presence of small numbers of hard faults in protected structures. In our study of the latency-sensitive core, we have shown that SRAM-based designs have low latencies

that end up providing less benefit to a throughput-oriented core and workload than a better-fitted data cache composed of DRAM. The move from a high-power, low-latency technology to a lower-power, high-latency technology allows us to increase L1 data cache capacity, which is a net benefit for the throughput-oriented core.

Dedication

To the Doctors Bower that precede me, those that may follow, and for Ronnie.

Table of Contents

Abstract iv

List of Figures xiii

List of Tables xv

Acknowledgements xvi

Copyright Acknowledgements xviii

1 Introduction 1

1.1 Single Core Trends in Latency Sensitive Applications 3

1.2 Throughput-Oriented CMP Trends 4

1.3 Thesis Statement and Contributions 6

1.4 Thesis Outline 7

2 Fine-Grained Hard Fault Tolerance in Single Core Applications 8

2.1 Fault Tolerance Background 10

2.1.1 Hard Faults in Submicron CMOS Technology 11

2.1.1.1 Fault Models 11

2.1.1.2 Underlying Physical Phenomena 11

2.1.2 Existing Fault Tolerance Techniques 14

2.2 Self-Repairing Arrays 16

2.2.1 Microprocessor Array Structures 18

2.2.1.1 Reorder Buffer 18

2.2.1.2 Branch History Table 19

2.2.2 Design Space 20

2.2.3 SRAS-CheckRow (SRAS-CR) 21

2.2.3.1 Detection and Diagnosis 21

2.2.3.2 Recovery 23

2.2.3.3 Mapping Out Faulty Sub-arrays 23

2.2.3.4 ROB Remapper 24

2.2.3.5 BHT Remapper 25

2.2.4 SRAS-CR Costs 27

2.2.5 Limitations of SRAS-CR 28

2.2.6 SRAS-EDC: Self-Repair Design Without DIVA Backstop 29

2.2.6.1 Detection and Diagnosis 31

2.2.6.2 Recovery 34

2.2.6.3 Remapping 34

2.2.6.4 SRAS-EDC Costs 35

2.2.6.5 Limitations of SRAS-EDC 35

2.2.7 Applicability of SRAS to Specific Structures 35

2.2.7.1 Instruction Buffer 35

2.2.7.2 Instruction Scheduling Window 36

2.2.7.3 Load-Store Queue 37

2.2.7.4 Branch History Table 38

2.2.7.5 Reorder Buffer 38

2.3 Online Diagnosis of Hard Faults in Microprocessors 38

2.3.1 Fault Diagnosis 39

2.3.2 A New Online Diagnosis Mechanism 40

2.3.2.1 Design Issues 41

2.3.2.2 Heuristics for Choosing Error Counter Values 43

2.3.2.3 Discussion 45

2.3.2.4 Alternative Design Options 47

2.3.3 Deconfiguring Faulty Components 49

2.3.4 Costs and Limitations 51

2.3.4.1 Hardware Costs 51

2.3.4.2 Limitations 51

2.4 Evaluation 53

2.4.1 Experimental Methodology and System Model 53

2.4.2 SRAS-CR and SRAS-EDC 56

2.4.2.1 Fault-Free Performance 56

2.4.2.2 Performance in Presence of Faults 57

2.4.2.3 Relative Performance Impact of Protecting Different Arrays 60

2.4.2.4 Implementation Costs 62

2.4.3 Online Diagnosis 63

2.4.3.1 Detection and Diagnosis of Hard Faults 64

2.4.3.2 Performance After Deconfiguring FDU 70

2.4.3.3 Performance with Just DIVA Recovery (But No Diagnosis) 71

2.4.4 Summary and Discussion of Results 77

2.4.5 Related Work 80

3 Extending DRAM Use to the Level 1 Data Cache in Throughput-Oriented CMPs 83

3.1 Experimental Methodology 85

3.2 Demands of Throughput-Oriented Workloads 88

3.2.1 Bandwidth Demands of the Throughput-Oriented Core 88

3.2.2 L1 Data Cache Latency Sensitivity 90

3.2.3 Throughput-Oriented Cache Demand Summary 92

3.3 Cache Building Block Technology Alternatives 92

3.3.1 Build a Better SRAM Cell 93

3.3.2 Embedded DRAM as an Alternative to SRAM 94

3.3.3 Hybrid Caches 94

3.4 Experimental Cache Design Space Exploration 95

3.4.1 L1 Data Cache Evaluation 96

3.4.1.1 Bandwidth and Power 96

3.4.1.2 Cache Area 100

3.4.1.3 Cache Latency 101

3.4.1.4 Summary of L1 Data Cache Evaluation 102

3.4.2 L2 Cache Evaluation 102

3.4.3 Putting it All Together: Evaluation of a Throughput-Oriented Cache 104

3.4.3.1 Improvements in the L1 Data Cache 105

3.4.3.2 Opportunities in the L2 Cache 107

3.5 Related Work 108

4 Summary and Conclusions 109

4.1 Summary of Results 110

4.2 Conclusions 112

References 114

Biography 123

List of Figures

Figure 2-1. Oxide Breakdown Process and its Circuit Level Implications 12

Figure 2-2. Broader Impact of OBD in the Circuit 12

Figure 2-3. Array Remapping 24

Figure 2-4. Deconfiguration of Entries in a Circular Buffer (e.g., Reorder Buffer) 24

Figure 2-5. Deconfiguration of Entries in a Tabular Structure (e.g., Reservation Station) 24

Figure 2-6. Datapath Design with SRAS-EDC 30

Figure 2-7. Fault-Free Runtime 58

Figure 2-8. Runtime with Hard Faults Injected into the Reorder Buffer 59

Figure 2-9. Impact on Runtime of Hard Faults on Other Array Structures 61

Figure 2-10. Error-Free Performance (SPECfp and SPECint) for Each of the Three Evaluated Processor Configurations 64

Figure 2-11. Hard Fault Diagnosis Latency, Averaged Over All Benchmarks, for Narrow Configuration 65

Figure 2-12. Performance Impact of Losing One Component to a Hard Fault for Each of the Three Evaluated Processor Configurations 67

Figure 2-13. Performance of DIVA-Only Correction for Combinational Logic Units (SPECint) 72

Figure 2-14. Performance of DIVA-Only Correction for Combinational Logic (SPECfp) 73

Figure 2-15. Performance of DIVA-Only Correction for Array Logic Units (SPECint) 74

Figure 2-16. Performance of DIVA-Only Correction for Array Logic Units (SPECfp) 75

Figure 3-1. Dynamic Instruction Stream Memory Instruction Mix 89

Figure 3-2. Throughput-Oriented Core Bandwidth Demands on L1 Data Cache 89

Figure 3-3. Apache Web Server and SPECjbb Normalized Throughput on Out-of-Order, Multithreaded Cores with Varying L1 Data Cache Latency 91

Figure 3-4. Apache Web Server and SPECjbb Normalized Throughput on In-Order, Multithreaded Cores with Varying L1 Data Cache Latency 91

Figure 3-6. Power Comparison of SRAM, 1T1C DRAM, and Hybrid [77] L1 Data Cache Designs 97

Figure 3-5. L1 Data Cache Bandwidth for Selected Designs as Core Frequency is Varied 97

Figure 3-8. Latency of SRAM and DRAM-Based L1 Data Cache Implementations 100

Figure 3-7. L1 Data Cache Area for Energy-Delay Optimized Designs Implemented with SRAM, DRAM, or Hybrid Cells 100

Figure 3-9. Area and Power Characteristics of Large L2 Cache Configurations for Large-Scale CMPs 104

Figure 3-10. Latency to Access SRAM and DRAM L2 Caches at 750MHz Operating Frequency 104

Figure 3-11. Speedup by 8-Thread Configurations Over 2-Thread Core with 16 KB, 1-Cycle L1 Data Cache at L1 Data Cache Capacities From 16KB to 1MB and 1, 2, and 4-Cycle Latencies 106

List of Tables

Table 2-1. Fault Tolerance Techniques: Design Points and Limitations 17

Table 2-2. Error Counter Thresholds 45

Table 2-3. SRAS Target System Parameters 54

Table 2-4. Parameters of Target Systems for Online Diagnosis Evaluation 55

Table 2-5. Number of Diagnoses Needed to Identify Correct Failing Unit 68

Table 3-1. Processor Configuration for L1 Data Cache Latency Sensitivity Study 86

Table 3-2. L1 Data Cache Configurations Explored 87

Table 3-3. L2 Cache Configurations Explored 103

Table 3-4. Cache Power, in Milliwatts at Maximum Throughput 105

Acknowledgements

There are many people to whom I owe a debt of gratitude for their support of my work. I would like to acknowledge that support here. First and foremost, I am grateful for the support of my parents, who have provided me with an upbringing that values education and who have backed my efforts fully. I am also thankful for the support of my extended family throughout my long journey. I am most appreciative of the patience of my wife, son, and dogs, all of whom have had to often come in second place as I have worked toward this goal. They share this accomplishment, though their contributions are not directly written here.

I am appreciative of the patience of my advisor, Dan Sorin, and my IBM advisor, Steve Hunter, both of whom have helped me achieve this goal, despite the many challenges that presented themselves along the way. Gauging the abilities and motivators of another is a skill that Dan has acquired new depth in throughout his tenure advising this dissertation, and I am grateful that he has endured through the process to my definition of success.

I have many IBM colleagues to thank, but those that have been the most instrumental to my success are Celia Schreiber, Bill Ott, and Steve Levesque. Celia and Bill supported me at the beginning and Steve has supported me at the end, when time has been hard to come by. The understanding and commitment on the part of the company, as personified by the support of these three has made this accomplishment possible.

To my extended Durham community, I am also thankful. My Duke Architecture office mates, Dr. Tong Li, Dr. Albert Meixner, and Dr. Anita Lungu all precede me in the alumni ranks and their example has helped to guide me to completion. My village of coffee shop denizens has been supportive in ways that they could not have possibly recognized the value of at the time, but their constant presence has been part of my success here.

My final note of gratitude is due to Diane Riggs, my personal guide through the administrative maze that is the Duke Graduate School and Computer Science Department. More so than for a typical student, Diane has been a constant shepherd of my progress and guide through the process of completing my degree requirements. Without her help, I would not be writing these words now. I wish her the best in her life after Duke and can only hope that those that may follow have as good a guide as Diane has been.

Copyright Acknowledgements

Content found in Chapter 2 has been previously published and is republished here with permission from ACM and IEEE. The following copyright notices apply to the applicable content in Chapter 2.

©ACM, 2007. This is the author's version of the work. It is posted here by permission of ACM for your personal use. Not for redistribution. The definitive version was published in ACM Transactions on Architecture and Code Optimization (TACO), {4, 2, (June 2007)} http://doi.acm.org/10.1145/1250727.1250728

This material is posted here with permission of the IEEE. Such permission of the IEEE does not in any way imply IEEE endorsement of any of Duke University's products or services. Internal or personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution must be obtained from the IEEE by writing to pubs-permissions@ieee.org. By choosing to view this material, you agree to all provisions of the copyright laws protecting it.

1 Introduction

As computer architects, we continue to benefit from the decades-long progression of Moore's Law [46]. Increasing device densities into deep sub-micron feature geometries have enabled further performance gains in latency-sensitive single-core applications, and have also ushered in the generation of throughput-oriented computing. Chip multiprocessors (CMPs) now are ubiquitous in personal computers and servers. With the technological advances that enable these increased densities comes a set of additional challenges.

First, with smaller device geometries and lower operating voltages, devices are more sensitive to many of the fault causing effects. These effects stem from manufacturing defects, such as particle contamination or process variation, and from wearout effects due to gate oxide breakdown and electromigration [13, 30, 51, 68]. Increased sensitivity presents additional challenges in process yield at the factory. It also makes parts more sensitive to progressive field wearout effects [8] and the phenomena that incur transient faults.

This increased sensitivity has caused device reliability in high-performance to become a greater concern with smaller geometries. Techniques that help to maintain or improve manufacturing yields, extend part lifetimes, and provide graceful degradation in the presence of common faults in critical structures are increasingly important in these designs as they incorporate ever-more transistors and per-transistor fault rates fail to keep pace with these increases.

Next, while we continue to lower operating voltages to reduce power consumption, we are approaching a point where the ability to scale voltage further is limited by physical limits. Until recent technology generations, scaling of voltage provided an effective means to keep chip power from growing as we add more devices to the same area. With this technique losing its ability to keep pace with further scaling, we now face a challenge to reduce power via other means, such as

reducing the average activity factor of transistors on the part. Reduction of threshold voltage has also brought with it an increase in the leakage component of the power consumption by traditional device designs such as the six-transistor (6T) SRAM cell commonly employed as a building block for on-chip storage, both within the core and in the cache hierarchy.

Finally, as densities have increased, basic cooling limits of the total microprocessor package have not. This is due to the fact that maximum die area has remained relatively fixed at roughly 400 mm^2 and advances in heatsink technology have not provided significant additional gains in heat dissipation from the package. With a physical cap on power dissipation, power has become a first-class design constraint.

These challenges present a new set of boundaries that constrain the microarchitect in extracting additional value from the ever-increasing transistor budget. At the same time, efforts to improve single-threaded performance have been thwarted by challenges with scalability of performance-critical structures in the core and the power challenges of increasing chip frequency. This has led the microarchitecture community away from techniques that seek to increase ILP in single-threaded applications. Instead, we are now seeing increased focus on optimizing energy-delay to achieve balanced throughput per unit power with reasonable latency in many applications, particularly in the commercial application space of the Internet.

As a result of these trends, we develop techniques to cope with these fundamental challenges in both latency-sensitive and throughput-oriented paradigms. In latency-sensitive core design space, we propose and evaluate fine-grained hard fault tolerance mechanisms within the core. In the throughput-oriented CMP design space, we study the demands of the throughput-oriented core and data cache alternatives that provide a better match to these demands than traditional cache designs.

The rest of this chapter is organized as follows. In Section 1.1, we discuss trends that motivate our research in the latency-sensitive core design space. In Section 1.2, we explore how these general trends impact the throughput-oriented core design space. Section 1.3 presents our thesis statement and the hypotheses that we test in this work. We conclude in Section 1.4 with an outline of the rest of the dissertation.

1.1 Single Core Trends in Latency Sensitive Applications

As technological trends continue to lead toward smaller device and wire dimensions in integrated circuits, the probability of hard (permanent) faults in microprocessors increases. These faults may be introduced during fabrication, as defects, or they may occur during the operational lifetime of the microprocessor. Well-known physical phenomena that lead to operational hard faults are gate oxide breakdown, electromigration, and thermal cycling. Microprocessors become more susceptible to all of these phenomena as device dimensions shrink [68], and the semiconductor industry's roadmap has identified both operational hard faults and fabrication defects (which we will collectively refer to as "hard faults") as critical challenges [28]. In the near future, it may no longer be a cost-effective strategy to discard a microprocessor with one or more hard faults, which is what, for the most part, we do today.

Traditional approaches to tolerating hard faults have masked them using macro-scale redundancy, such as triple modular redundancy (TMR). TMR is an effective approach, but it incurs a 200% overhead in terms of hardware and power consumption. There are some other, lightweight approaches that use marginal amounts of redundancy to protect specific portions of the microprocessor, such as the cache [49, 84], but none of these are comprehensive.

Our goal in this work is to create a microprocessor design that can tolerate hard faults without adding significant redundancy. The key observation, made also by previous research [64, 67, 69],

is that modern latency-sensitive, superscalar microprocessors, particularly those supporting simultaneously multithreading (SMT) [76], already contain significant amounts of redundancy for purposes of exploiting ILP and enhancing performance. We want to use this redundancy to mask hard faults, at the cost of a graceful degradation in performance for microprocessors with hard faults. In this work, we do not consider adding extra redundancy strictly for fault tolerance, because cost is such an important factor for commodity microprocessors. The viability of our approach depends only on whether, given a faulty microprocessor core, being able to use it with somewhat degraded performance provides any utility over having to discard it.

To achieve our goal, the microprocessor core must be able to do three things while it is running.

1) It must detect and correct errors caused by faults (both hard and transient).

2) It must diagnose where a hard fault is, at the granularity of the field deconfigurable unit (FDU).

3) It must deconfigure a faulty FDU in order to prevent its fault from being exercised.

While previous work in this area has explored aspects of this problem, none has developed an integrated solution. Some work has used deconfiguration to tolerate strictly fabrication defects and thus assumed pre-shipment testing instead of online error detection and diagnosis [64]. Other work has explored deconfiguration and has left detection and diagnosis as open problems [69]. We will discuss integrated design options for microprocessors that achieve all three of these goals in Chapter 2.

1.2 Throughput-Oriented CMP Trends

Multicore processors are now the standard commodity computing platform. Many researchers argue that Moore's law will lead to exponential increases in the number of cores per chip. In this scenario the memory system that serves these cores becomes a crucial system component in terms

of performance and power. As power and energy efficiency become first-class design constraints, the ability to increase clock frequencies as the primary means for improved performance becomes infeasible.

The response by system designers has been to place greater emphasis on exploiting coarse-grained, thread-level parallelism (TLP) by increasing the number of cores on a single chip. This new paradigm represents an opportunity to revisit single-chip designs. Specifically, for many server workloads, throughput is the primary performance metric. The realization that a more energy-efficient design point can be achieved for throughput-oriented computing led to a redesign of the cores. Recent chip multiprocessors (e.g. UltraSparc T2 [62]) have a large number of simpler low-power processor cores, often with multithreading for tolerating long latency events, instead of a small number of sophisticated high-power cores.

Although *core* microarchitectures have been re-designed for throughput-oriented computing, the *memory systems* are still tailored to the demands of high-performance, latency-centric cores. Instead, designs should seek to balance latency, bandwidth, and capacity for optimum throughput. The aim of the work we present in Chapter 3 is to identify and exploit mis-matches in the capabilities of the on-chip data cache and the throughput-oriented core. The first part of this work is to show that the low latency caches of the late single-core and early multicore era over-emphasize the criticality of latency for a throughput-oriented workload. With the magnitude and nature of the miss-match better understood, we explore ways in which we can trade over-provisioned attributes, such as latency, for attributes that will benefit the throughput-oriented CMP, namely additional cache capacity and power savings.

1.3 Thesis Statement and Contributions

With this thesis, our goal is to validate two primary hypotheses motivated by the high-level trends we have reviewed in this introduction: 1) fine-grained techniques for detecting, diagnosing, and tolerating hard faults in latency-sensitive cores can provide performance of a fault-free core in both fault-free and fault-present states at a fraction of the hardware and power costs of traditional coarse-grained fault-tolerance methods and 2) a throughput-oriented cache design, better matched to core demands, enables additional throughput gains over present designs under a fixed power budget.

In support of these hypotheses, we make three primary contributions:

1) Our first contribution is a fine-grained fault diagnosis and deconfiguration technique for array structures, such as the ROB, within the microprocessor core. We present and evaluate two variants of this technique. The first variant uses an existing fault detection and correction technique scoped to the processor core execution pipeline to ensure correct processor operation. The second variant integrates fault detection and correction into the array structure itself to provide a self-contained, fine-grained, fault detection, diagnosis, and repair technique.

2) In our second contribution, we develop a lightweight, fine-grained fault diagnosis mechanism for the processor core. In this work, we leverage the first contribution's methods to provide deconfiguration of faulty array elements. We additionally extend the scope of that work to include all pipeline circuitry from instruction-issue to retirement.

3) In our third and final contribution, we study the demands of the throughput-oriented core running a representative workload and then propose and evaluate an alternative data cache implementation that more closely matches the demands of the core. We then show that a better-

matched cache design can be exploited to provide improved throughput under a fixed power budget.

1.4 Thesis Outline

We begin our presentation of contributions in Chapter 2, with a presentation of our fine-grained fault tolerance and diagnosis techniques. In Chapter 3, we present our work on throughput-oriented cache design. The thesis concludes with a summary of contributions and conclusions in Chapter 4.

2 Fine-Grained Hard Fault Tolerance in Single Core Applications[1]

In this chapter we develop and evaluate techniques to provide fine-grained hard-fault tolerance in the high-performance microprocessor core. With large, complex core implementations, fine-grained techniques afford the designer with a way to provide graceful degradation of performance in the presence of small numbers of faults, even in critical structures. With the methods that we develop, our evaluation shows that performance losses can be mitigated to a point where utility of the part is retained for extended periods of operation after faults are encountered. The chapter begins with a definition of the fault models that we use and a discussion of existing techniques for providing fault tolerance in the microprocessor core.

We then discuss the design space for self-repairing microprocessor array structures, and we present two specific designs. Array structures include the reorder buffer, load-store queue, instruction queue, branch history table, etc. Our goal is to develop self-repairing arrays that enable autonomic execution. In both of our designs for self-repairing array structures (SRAS), spare rows are built into each array structure and are mapped in to replace faulty rows using a level of indirection. This approach is similar to how disks map out faulty sectors and how hard faults in DRAMs can be tolerated with schemes that map out faulty locations [19, 44, 59]. Our first design, SRAS-Check-Row (SRAS-CR), uses dedicated check rows to detect and diagnose hard faults. SRAS-CR relies upon DIVA [6] to recover from transient errors and errors due to hard faults that have not yet been classified as hard. Our second design, SRAS-EDC, uses error detecting codes (EDC) for error

1. This chapter contains previously published work that is covered by the following copyrights:

©2005 IEEE. Reprinted, with permission, from IEEE Transactions on Dependable and Secure Computing, Autonomic Microprocessor Execution via Self-Repairing Arrays, Fred A. Bower, Sule Ozev, and Daniel J. Sorin.

©ACM, 2007. This is the author's version of the work. It is posted here by permission of ACM for your personal use. Not for redistribution. The definitive version was published in ACM Transactions on Architecture and Code Optimization (TACO), {4, 2, (June 2007)} http://doi.acm.org/10.1145/1250727.1250728

detection/diagnosis, and it uses the pre-existing branch misprediction recovery mechanism to recover from transient errors and errors due to hard faults that have not yet been classified as hard. After a hard fault has been diagnosed and mapped out, neither SRAS-CR nor SRAS-EDC incurs a performance penalty due to that fault, unlike lightweight schemes that incur a costly recovery for every manifestation of a hard fault.

Our experimental results show that SRAS-EDC adds some performance overhead in the fault-free case, but that both SRAS-CR and SRAS-EDC mask hard faults (a) without the hardware costs of high-level redundancy (e.g., IBM mainframes [66]) and (b) without the per-error performance penalty of existing low-cost techniques (e.g., DIVA). When hard faults are present in arrays, due to operational faults or fabrication defects, then our SRAS schemes outperform low-cost techniques that require a pipeline recovery per error. Given the increasing frequencies of fabrication defects and operational hard faults, the likelihood of wanting to be able to operate correctly with one or more hard faults makes array self-repair appealing.

With our two SRAS implementations defined, we expand our scope to develop an online, fine-grained fault diagnosis and deconfiguration mechanism for the microprocessor core. In this work, we utilize SRARS-style methods for deconfiguration of faulty array structures. We also extend our ability to diagnose faults to include functional units and data paths within the processor pipeline. Our experimental results show that our new diagnosis mechanism quickly and accurately diagnoses hard faults. Moreover, our reliable microprocessor can function quite capably in the presence of hard faults, despite not using redundancy beyond that which is already available in a modern microprocessor. This technique can turn otherwise useless microprocessors into microprocessors that can function at a gracefully degraded level of performance. This capability can improve reliability by tolerating operational hard faults. We can improve yield by shipping micro-

processors with defects that we have tolerated—it is as if they are regular microprocessors that will get "binned" into a lower performance bin. Although binning is typically by clock frequency, recent proposals have suggested more general performance binning [64]. As long as these bins are not so low-performing as to be useless, then our improvement in yield is a benefit. Our scheme also vastly outperforms a system with only DIVA or a comparable recovery-based scheme, since the performance cost of recoveries is quite high for hard faults that get exercised frequently; moreover, our scheme can tolerate a hard fault in a DIVA checker.

The rest of this chapter is organized as follows. Section 2.1 provides background on our hard fault model. The underlying physical phenomena that lead to hard faults are discussed in some detail to familiarize the reader with these mechanisms as well as to further motivate the case for providing hard-fault tolerance in coming microprocessor core designs. Section 2.2 presents SRAS-CR and SRAS-EDC in detail, explaining the mechanisms, how they operate, and their limitations and advantages in the application of providing hard-fault tolerance to microprocessor core array structures. In Section 2.3, we present our fine-grained diagnosis and deconfiguration framework for high-performance microprocessor cores. We conclude this chapter in Section 2.4, with the experimental evaluation of these techniques.

2.1 Fault Tolerance Background

In this section, we first define terminology around fault tolerance that we will utilize throughout the rest of the chapter. We also present the historical progression of designs that has led up to this point and motivated the work in this chapter.

2.1.1 Hard Faults in Submicron CMOS Technology

We start with a presentation of existing high-level models for hard faults (Section 2.1.1.1) and then we delve into the underlying physical phenomena that cause hard faults (Section 2.1.1.2). In this process, we show that existing fault models are applicable to the physical faults that we consider in this work

2.1.1.1 Fault Models

To facilitate fault tolerant design and testing for physical faults that lead to errors at the circuit level, several structural fault models have been developed for logic circuits and storage components over the past few decades [1]. The *stuck-at fault model* is the most commonly used model in VLSI testing and fault tolerance schemes. In this model, a physical defect manifests itself as a signal consistently having a certain value (either zero or one) independent of the input. For example, an unintended short circuit between the two inputs of an XOR gate results in a stuck-at-zero fault at the output signal. The *coupling fault model*—in which a write to a certain memory location always prompts a write to a neighboring location or locations—has been defined for storage components [18]. The recently defined *transition fault model* represents a slow charging or discharging of a circuit node [53, 60, 75]. This delay can cause incorrect logic values to be latched.

Next, in Section 2.1.1.2, we see that stuck-at and coupling fault models will be sufficient for the hard faults that we consider.

2.1.1.2 Underlying Physical Phenomena

The reliability of electronic devices under discrete environmental stress, such as radiation [70], and continuous functional stress due to the applied electric field [11, 57, 72] has been a topic of vast research since the early days of semiconductor manufacturing. Extensive research has been conducted on the failure-causing physical phenomena, such as electromigration [11, 34, 72] and

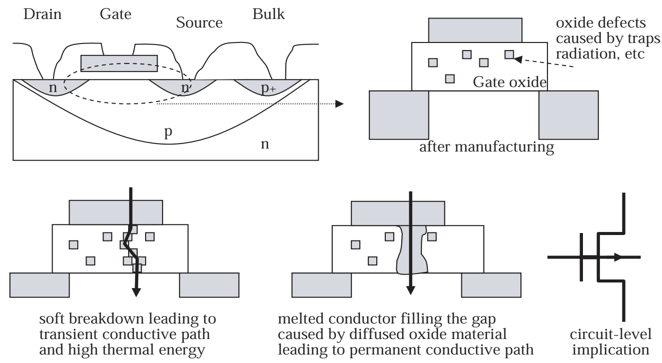


Figure 2-1. Oxide Breakdown Process and its Circuit Level Implications

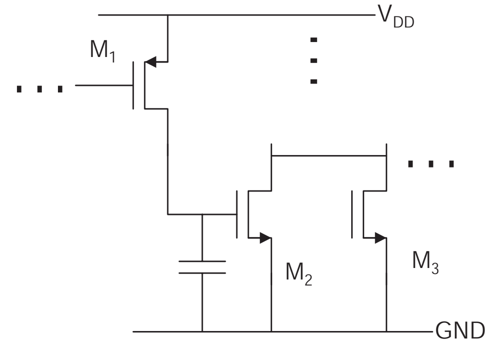


Figure 2-2. Broader Impact of OBD in the Circuit

transistor gate oxide breakdown (OBD) [23]. There have been several recent studies of operational hard faults [30, 68], that is, hard faults that occur over the lifetime of the microprocessor. Srinivasan et al. [68] determine that electromigration and gate oxide breakdown are likely to be the two dominant phenomena that cause operational hard faults, thus we focus on them in developing our fault models in this work. Electromigration results in highly resistive interconnects or contacts and eventually leads to open circuits. Such defects are typically modeled as transition faults during manufacturing testing, but they become stuck-at faults during operation due to their progressive nature.

Gate oxide breakdown (OBD) results in the malfunction of a single transistor due to the creation of a highly conductive path between its gate and its bulk. As illustrated in Figure 2-1, a newly manufactured oxide contains inherent electron traps due to imperfections in the fabrication process. Over the lifetime of the device, the number of such traps increases due to electric field stress and electron tunneling. At some point, the electron traps may line up and constitute a conductive path between the gate and the bulk of the device. The onset of this phenomenon is called a soft breakdown (SBD). OBD increases switching delay. It can lead to delay faults that manifest themselves as bit flips [16]. Initially, the conductive path may be transient since the high heat

caused by high current density may relocate some of the traps. However, after several SBD incidents, the oxide layer diffuses and highly conductive melted metal fills the void and solidifies into a consistent path. This phenomenon is called hard breakdown (HBD). Similar to the electromigration case, the initial circuit level manifestation of SBD is a transition fault, whereas the effect of the subsequent HBD is a stuck-at fault. OBD defects are potentially more dangerous than electromigration defects due to the consistent path between a charged node and ground or supply. For the circuit illustrated in Figure 2-2, an OBD defect in transistor M2 forms a conductive path between the drain of M1 (and the gate of M2) and the ground node. As long as the logic value at the gate of M1 is LOW, there is a sustained resistive path from the supply to the ground, resulting in sustained current flow through transistor M1. Since the resistance of this transistor for a LOW input is typically small, the current can be large, potentially damaging M1 or causing regional drops in the supply. Thus, detection and isolation of memory locations with OBD defects is essential for the operational health of computing devices.

Both the electromigration and OBD defects are progressive in nature. The mean time to failure (MTTF) for both defects depends on the thickness and the initial health of the structure. Reported laboratory data on OBD indicates that MTTF is on the order of four million seconds (around 46 days) for 15Å gate oxides under constant stress of 2.1V [40] (scaling the supply voltage down to 1.0V, we can estimate the MTTF for this oxide thickness to be 375 days). However, MTTF also depends heavily on the number and location of initial traps within the oxide; thus it can be much shorter for some transistors with inherent weaknesses. A similar analysis can be made for electromigration defects [12].

In the early stages of the progression of both electromigration defects and OBD defects, bit errors only occur if the defects are sequentially excited. However, in later stages, both defects

resemble stuck-at faults. Moreover, in addition to affecting the output node to which the defective transistor is connected, the OBD defects may result in coupling faults due to their current driving nature. Thus, in our evaluation experiments, we inject stuck-at faults and coupling faults since they correspond to the manifestations of electromigration and OBD defects.

Defects introduced during chip fabrication are another source of hard faults. Their causes differ from those of operational hard faults, but they often manifest themselves in a similar fashion. For example, a fabrication defect could result in a discontinuity in a wire, which is equivalent to the situation in which electromigration leads to an open circuit. A fabrication defect could also lead to the growth of an insufficiently thick gate oxide, which is functionally equivalent to OBD. The impact of technology trends on fabrication defects is less clear than it is for operational faults. In general, though, smaller wire and device dimensions are more prone to defects, since the margin for error is smaller.

2.1.2 Existing Fault Tolerance Techniques

A canonical design for autonomic operation is the IBM mainframe [66]. Mainframes not only have redundant processors, but they also incorporate redundancy within the processor in order to seamlessly tolerate hard faults. The IBM G5 microprocessor, for example, has redundant units for fetch/decode and for instruction execution. Some other traditional fault-tolerant computers, such as the Stratus [82] and the Tandem S2 [31], simply replicate entire processors. While these systems all provide excellent reliability, such heavyweight redundancy incurs significant costs in terms of hardware and power consumption.

As a low cost and low power alternative to heavyweight redundancy, DIVA [6] dynamically verifies an aggressive microprocessor core with a simple, provably correct checker core. DIVA sacrifices some amount of reliability in order to greatly reduce these costs. DIVA's small amount

of redundancy uses far less power than mainframe redundancy, but it incurs significant performance and energy penalties for each error that it must correct. Each error detected and corrected by the checker core triggers a pipeline flush of the aggressive core. Since DIVA was designed primarily for soft faults (not the hard faults we target), these flushes are not a performance problem. However, permanent faults in frequently accessed structures, such as the reorder buffer, will frequently manifest themselves as errors and will thus greatly degrade performance. Researchers have also proposed using redundant threads to achieve lightweight redundancy, primarily for soft faults. Of these schemes, the ones that perform recovery as well as error detection include AR-SMT [58], Slipstream [71], and SRTR [78]. All of these schemes share the same drawback as DIVA, with respect to hard faults, since they incur a pipeline squash (and its corresponding performance and energy penalty) every time a hard fault manifests itself. Redundant thread schemes, unlike DIVA, may not be able to guarantee forward progress in the presence of hard faults.

One option for array structures is to protect them with error correcting codes (ECC), as in IBM mainframes [66]. Combining ECC for arrays with DIVA avoids costly DIVA recoveries. However, ECC protection of arrays is on the critical path for array access (both read and write). Current ECC implementations can calculate ECC on a representative datum in 4 cycles on a 2 GHz Itanium2 [81]. Since ECC must be calculated on the microprocessor's critical path, a 4-cycle penalty per ECC calculation results in highly-degraded performance, even in the fault-free case. This lost performance makes ECC inappropriate for application in the timing-critical microprocessor pipeline.

With the advent of chip multi-processing (CMP) in commodity microprocessor designs, another hard-fault tolerance option is to disable any core that is detected to have a hard fault. While this works, we seek to provide a more cost-effective option than to lose 1/Nth (for an N-core design) of the chip's capacity for each hard fault that is detected. Aggarwal et al. [2, 3] extend this

idea to include other shared CMP structures, such as memory controllers and on-chip busses. In their presented methodology, a designer can add on-chip wiring complexity and mulitplexor delay to gain the ability to route around faulty shared components. Shivakumar et al. [64] propose a more cost-effective, fine-grained solution that utilizes inherent redundancy in CMP and SMT designs. This work is limited to manufacturing-time detection (i.e., testing) and deconfiguration. The methods that we present in this work, SRAS-CR, SRAS-EDC, and our new online fault diagnosis mechanism, provide a means for both manufacturing-time and in-situ operational detection and deconfiguration of sub-units within the microprocessor core, giving the designer additional options in designing for hard-fault tolerance.

Table 2-1 summarizes all of these techniques, including our SRAS-CR, SRAS-EDC, and online diagnosis (labeled Microarchitectural Redundancy Exploitation) designs. Included are the original fault-tolerance targets of the techniques (soft, hard, or design), and notes on the limitations of using these in a commodity microprocessor design. Note that our online fault isolation design extends previous work [2, 3, 64] to provide a finer-granularity of redundancy exploitation within the core.

As can be seen in the table, each technique has certain advantages and certain disadvantages. The characteristics of a given technique make it more or less appropriate for application to a given design space.

2.2 Self-Repairing Arrays

Technology and microprocessor architecture trends are leading towards larger array structures within microprocessors. These structures include the instruction queue, reorder buffer (ROB), register file, reservation stations, register map table, branch history table (BHT), etc. These structures

Table 2-1. Fault Tolerance Techniques: Design Points and Limitations

Technique	Primary Fault Target(s)	Limitations of Use for Hard-Fault Tolerance In the Microprocessor Core
DIVA	Soft, Design	Excessive performance penalty for frequent pipeline flushes due to faults in frequently-accessed structures
Redundant Multithreading	Soft	May be subject to livelock Excessive performance penalty for frequent pipeline flushes due to faults in frequently-accessed structures
Triple Modular Redundancy (TMR)	Soft, Hard	Over 3x cost in terms of die area and power consumption over unprotected core design point
CMP Core Sparing	Hard	High performance penalty per hard fault (1 of *N* cores) in designs where *N* is relatively small
ECC	Soft, Hard	Adds excessive latency to critical path of microprocessor Only localized fault tolerance
Microarchitectural Redundancy Exploitation	Hard	Manufacturing-time only, as described in [64], coarse-grained in [2, 3], extended in this work to include faults in the field at a finer granularity within the core Only localized fault tolerance
SRAS-CR	Hard	Requires fault detection mechanism to trigger hard-fault tolerance Only localized fault tolerance
SRAS-EDC	Hard	Adds latency to fault-free operation of microprocessor Only localized fault tolerance

are the single-largest consumer of microprocessor core die area, comprising up to 33% of the area of microprocessor core (i.e., not including caches) in recent microprocessor designs [64]. We would like to protect these structures from hard faults as the probability of hard faults continues to increase, but we cannot afford to fully replicate these structures. Thus, our SRAS schemes protect array structures in a fashion similar to the way in which existing on-line (dynamic) techniques protect large memory storage structures. The basic idea is to use a level of indirection to map out faulty portions of the structure. Especially as structures grow larger, the probability of a hard fault within them increases. Disk sizes, for example, long ago reached the point at which hard faults were expected and had to be tolerated. Whole disk failures were addressed by RAID [52]. For disk faults that did not incapacitate the entire disk, the solution was to map out faulty portions at the

sector granularity. Thus, a faulty disk could continue to operate correctly in the presence of hard faults. Similar approaches have been developed for DRAM main memory. Whole chip failures are tolerated by chipkill memory and RAID-M [22, 27], and partial failures are tolerated with schemes that map out faulty locations [19, 44, 59]. For SRAM caches, techniques have been developed to map out defective locations during fabrication [84] and, more recently, during execution [49]. While providing insight for the use of spare memory locations for repair, direct application of the aforementioned methods to array structures within the processor bears little hope due to the performance criticality within microprocessors.

In the rest of this section, we discuss the arrays that we will protect (Section 2.2.1), and we present the design space for self-repairing arrays (Section 2.2.2). We then present two specific implementations (Section 2.2.3 and Section 2.2.6).

2.2.1 Microprocessor Array Structures

We can classify array structures within the microprocessor core into two categories: non-addressable buffers for which the data location is determined at the time of access, and randomly addressable tables for which the data location is determined before access. In order to allow timing efficient implementation of the repair logic, we exploit these distinct features of each type of array structures. Without loss of generality, we focus the discussion of SRAS on one specific array structure from each of the two categories: the reorder buffer (ROB) and the branch history table (BHT). The ROB and BHT are representative of the kinds of array structures found in modern microprocessors, and thus the arguments and results here apply broadly.

2.2.1.1 Reorder Buffer

The ROB is a circular buffer that is used in dynamically scheduled (a.k.a. “out-of-order”) processors to implement precise exceptions by ensuring that instructions are committed in program

order. There is an entry in the ROB for each in-flight instruction, and there are pointers to the head and tail entries in the ROB. An entry is added to the tail of the ROB once it has been decoded and is ready to be scheduled. An entry is removed from the head of the ROB when it is ready to be committed. We focus on processors that perform implicit register renaming with reservation stations—such as the Intel PentiumPro, IBM PowerPC, and AMD K6—in which an ROB entry contains the physical register tags for the destination register and the data result of the instruction. When an instruction commits from the head of the ROB, the data in the head entry is written to the destination register. Alternative ROB designs exist, in which ROB entries do not hold the data results of completed instructions (data is instead held in the physical registers). Designing SRAS for these alternative designs is straightforward and actually simpler (but not discussed in this work).

ROB sizes are on the order of 32-128 entries, which is large enough to have a non-negligible probability of a hard fault. The ROB has a high architectural vulnerability factor [48], in that a fault in an entry is likely to cause an incorrect execution. A fault in an ROB entry is not guaranteed to cause an incorrect execution for its instruction, though, since the fault might not change the data (i.e., logical masking) or the ROB entry might correspond to a squashed instruction (i.e., functional masking).

2.2.1.2 Branch History Table

The BHT is a table that is accessed during branch prediction. Common two-level branch predictor designs [83] use some combination of the branch program counter (PC) and the branch history register (BHR) to index into a BHT. The BHR is a k-bit shift register that contains the results of the past k branches. The indexed BHT entry contains the prediction (i.e., taken or not taken, but not the destination). A typical BHT entry is a 2-bit saturating counter [65] that is incremented

(decremented) when the corresponding branch is taken (not taken). A BHT value of 00 or 01 (10 or 11) is interpreted as a not-taken (taken) prediction.

BHRs and/or BHTs can be either local (one per branch PC), global (shared across all branch PCs), or shared (by sets of branch PCs). In this paper, we focus on the gshare two-level predictor [45], in which the BHT is indexed by the exclusive-OR of the branch PC and a global BHR. Since the BHT is a table, our remapper implementation for it is fairly similar to the logical abstraction presented earlier. The BHT has an architectural vulnerability factor of zero, in that no fault in it can ever lead to incorrect execution. However, a BHT fault can lead to incorrect branch predictions, which can degrade performance.

2.2.2 Design Space

Self-repairing arrays require three features, and the designs of each collectively comprise the design space:

- Detection of errors and diagnosis of faults

How does the hardware detect an error in an array, and then how does it isolate which part of the array is faulty? While there are several schemes for dynamically verifying microprocessor execution as a whole [6, 55, 58], they sacrifice diagnosis capability in order to not degrade performance.

- Recovery from errors

How does the hardware recover from an error such that it can ensure that the error does not propagate corrupted data into committed architectural state? The most basic option for recovery is to halt the system when an error is detected (fail-stop), thereby protecting system state from being corrupted, at the cost of more downtime and thus less availability. Other alternatives exist, such as using the microprocessor's branch misprediction recovery mechanism.

- Mapping out faulty sub-arrays

Once the faulty sub-array (e.g., row, column) has been diagnosed, how does the hardware map it out and thus avoid future manifestations of this fault? The design choices for this aspect mainly involve the granularity of mapping, e.g., row, column, or even the whole array. Another design decision is the number of spares to provide. These design decisions may be influenced by the array's position in the microprocessor pipeline, particularly if accessing the array is on the critical path and performance is thus crucial.

There are numerous design decisions for each of these three aspects, but the decisions for each aspect are not completely independent. For example, ECC protection of arrays would serve as the detection and recovery mechanism, and it does not require remapping, provided that the errors do not exceed the correction abilities of the chosen correction code.

The design decisions, particularly for the recovery mechanism, also determine which array structures can be protected. For example, since SRAS-EDC uses the misprediction recovery mechanism, it thus cannot tolerate errors in the recovery state (i.e., committed architectural state, such as the register file or condition codes).

2.2.3 SRAS-CheckRow (SRAS-CR)

The first SRAS design that we present, SRAS-CheckRows (SRAS-CR) uses dedicated check rows to detect and diagnose errors in array rows. SRAS-CR protects each array structure in isolation, i.e., the decision to protect an array with SRAS does not affect the decision to protect any other array. We will see in Section 2.2.6 that SRAS-EDC differs in that it is an integrated approach for protecting multiple arrays.

2.2.3.1 Detection and Diagnosis

SRAS-CR uses DIVA for end-to-end error detection and correction. However, DIVA cannot isolate the row or even the structure that is faulty. Thus, SRAS-CR combines DIVA with a simple scheme for detecting row errors and diagnosing which row is faulty. SRAS-CR adds a handful of check rows (some are spares, which are used to avoid a single point of failure) to each structure we wish to protect. For buffer structures such as the ROB, each time an entry is allocated, initialization data is written to both the entry and the check row. This initialization data consists of the available target data for the entry (for example, the source and destination register tags for an ROB entry) and pseudo-random data for the parts of the entry that will be written later (for example, the actual result value for an ROB entry). Where pseudo-random data is needed, the tick counter is used, with appropriate scaling to provide the proper number of bits to fully populate the entry. For tables, every write to a location will have a mirrored write to the structure's affiliated check row. Any partial write to a row must be implemented as a read-modify-write (RMW) action in order to support SRAS-CR checking. The issue here is that the check row and array entry to be checked must have identical data written into their contents in order for a meaningful comparison to be made. Immediately after the two writes, both locations are read and their data are compared (all off the critical path of execution). If the data differ, then one of the rows is faulty. Several options exist for determining which one is faulty, and we will explain a simple one after we first describe the mechanism we exploit for distinguishing hard faults from soft faults. SRAS-CR maintains small saturating counters for each row, which are periodically reset, and a counter value above a threshold identifies a hard fault. Now, to determine if the operational row or the check row is faulty, we can simply increment both of their counters in the case of a mismatch in their values, as long as we initially set the threshold for check row counters to be much higher than that for operational rows.

Detection and diagnosis is the same for both tables and buffers. While we logically need only k check rows in a k-way superscalar processor to detect and diagnose faults, the SRAS-CR imple-

mentation may necessitate having even more check rows. Having only *k* check rows could lead to an unreasonably long delay to transfer the data along wires from one end of the array to the other. Wire delays are already a problem in multi-GHz microprocessors—for example, the Intel Pentium4 has multiple pipeline stages allocated strictly to wire delay—and we cannot ignore them in our design. A simple option is to divide the array into sub-arrays, each of which has *k* check rows.

2.2.3.2 Recovery

If an error is detected, but the hard fault threshold has not yet been reached, then the fault is considered to be transient and it is tolerated with a DIVA recovery and its associated performance penalty. If the detected error raises the counter to the hard fault threshold, then DIVA also tolerates this fault, but the system then repairs itself so as to prevent this hard fault from being exercised again.

2.2.3.3 Mapping Out Faulty Sub-arrays

We logically add a level of indirection that can map out faulty rows in microprocessor array structures, as shown in Figure 2-3. The remapper serves as the interface between the array and the rest of the microprocessor.

The repair actions taken depend on whether the faulty row is a non-check row or a check row. If it is a non-check row, then it can be immediately mapped out and a spare row can be mapped in to take its place. The spare row can get the correct data from the check row. If the faulty row is a check row, then SRAS-CR maps in a spare check row.

While remapping with a level of indirection is straightforward in the abstract, implementing it in a high performance microprocessor pipeline requires careful consideration. We now present remapper implementations for the ROB and BHT.

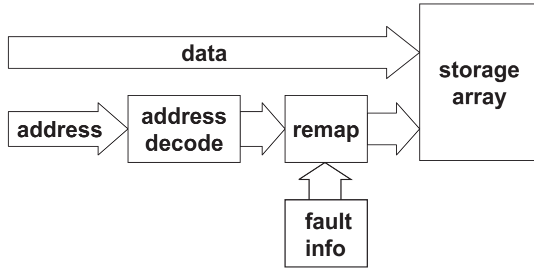


Figure 2-3. Array Remapping

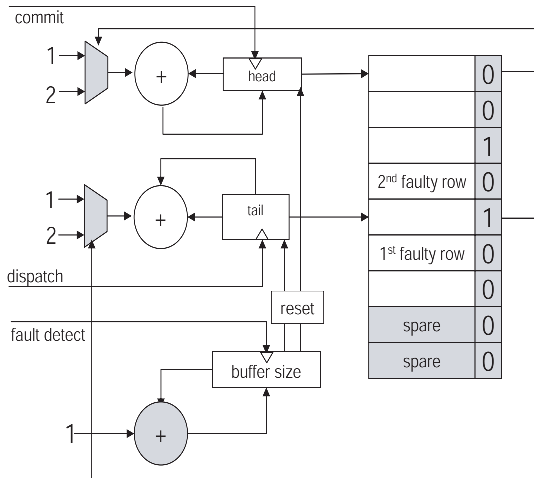


Figure 2-4. Deconfiguration of Entries in a Circular Buffer (e.g., Reorder Buffer)
Shading indicates hardware added for entry deconfiguration purposes.

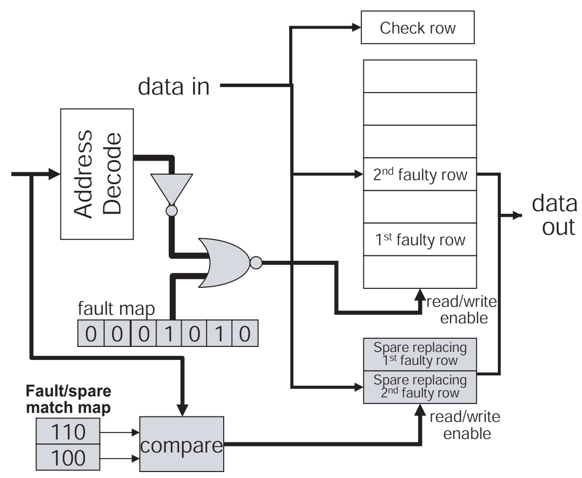


Figure 2-5. Deconfiguration of Entries in a Tabular Structure (e.g., Reservation Station)
Shading indicates hardware added for entry deconfiguration purposes.

2.2.3.3.1 ROB Remapper

In buffer structures, as in the case of the ROB, the address of the data to be accessed is determined at the time of the access. Typically, two pointers are used to mark the head and the tail location of the active rows. When a new ROB entry is allocated, the tail pointer is advanced and the corresponding address becomes the physical address of the data. Similarly, when an entry is removed, the head pointer is advanced. Thus, the physical as well as logical address of the data is abstracted and all rows have the same functionality. The faulty row can easily be mapped out by modifying the pointer advancement logic when a hard fault is detected. Figure 2-4 illustrates the

implementation of the self-repair mechanism for buffers, with SRAS-CR hardware shaded in gray. SRAS-CR uses a shifted fault map bit-array to track faulty rows. If a row is determined to contain a hard fault, the faulty bit in the previous row is set to 1. The fault map is used by the pointer advancement circuit to determine how far the pointer needs to be advanced. Upon the reception of a dispatch signal, the pointer is advanced by one or two depending on whether the next row is fault or not. The shifted faulty row information enables the preprocessing of the pointer advance logic. Upon the reception of the commit signal, the head pointer is advanced in the same manner. Once the pointer is updated accordingly, reads and writes of the buffer entries proceed unmodified. Since the pre-processing for pointer advancement can be done off the critical path, the proposed modification does not impact the read or write access time.

In order to avoid a reduction in the effective buffer capacity due to hard faults, spare rows can be used. Since there is no need to replace the faulty row with any particular spare row, the detection of the faulty row prompts incrementing the total buffer capacity by one entry (by adding the spare) while maintaining the same effective capacity. SRAS-CR can tolerate as many hard faults as there are spares without any degradation of buffer performance. If the number of faulty rows exceeds the number of spare rows, then the effective buffer capacity is allowed to shrink, resulting in graceful degradation of the buffer performance. Assuming that adding one or two to the pointers does not dramatically change timing or power consumption, the only overhead of this repair mechanism is the small additional area taken by the fault map and the additional power consumed for pointer pre-processing, updating fault map entries, and updating the buffer size. Section 2.2.4 discusses the overall overhead of the complete SRAS-CR architecture in more detail.

2.2.3.3.2 BHT Remapper

In tables, the logical address of the data is determined by the program execution prior to accessing the data. Since rows do not have equal functionality in tables, a faulty row needs to be replaced by a specific spare row. In this case, we need a logical indirection to map out the faulty rows. This problem is quite similar to the memory repair problem, and many on-line repair mechanisms have been proposed [11, 21]. However, in microprocessor array structures, logic inserted into the critical path directly impacts performance, so we must implement a timing-efficient repair mechanism. In SRAS-CR, we distribute spare rows over sub-arrays of the table, and a spare can only replace a row within its own sub-array. This choice may make the use of spares inefficient for highly localized faults, but it enables timing efficient implementation of the repair logic, as shown in Figure 2-5. Once again, hardware for SRAS-CR is shown in gray.

Similar to the buffer case, we keep the fault map information in a table. However, we also use a fault/spare match map which contains information on which functional row each spare row is replacing. If a row is identified faulty and an unused spare is found to replace it, the faulty entry of the row is set to 1. In addition, the physical address of the faulty row is written into the corresponding entry of the fault/spare match map. In the example shown in Figure 2-5, we can see that the 1st spare is allocated to the 6th entry and the 2nd spare is allocated to the 4th entry, hence the 1 in the fault map at the 6th position of the 1st column and the 4th position of the 2nd column. The address decode logic, which is present in all tables, enables a row of the table to be read or written by generating the individual read/write enable signals for the table rows. During a read or write access, these signals are modified by the remap logic to generate the updated read/write enable signals for the table entries as well as the read/write enable signals for the spare entries. The remap logic consists of n inverters and n 2-input NOR gates, where n is the size of the subarray. To generate the read/write enable signals for the spare rows, $k\ log(n)$ 2-input XOR gates and the equivalent of k $log(n)$-input NOR gates (denoted by the compare block in Figure 2-5) are needed, where k is the

number of spares assigned to the subarray. Note that, the fault/spare match map will contain one more bit than the physical address of the table to indicate whether the spare rows are active or not. This bit is not shown in the figure to avoid confusion with the address value.

Assuming the compare logic can execute faster than the address decode logic, SRAS-CR will add two gate delays (one INV and one NOR gate delay) to the table access time. Since the additional level of indirection for accessing the physical table entries is on the critical path, this additional time cannot be ignored. In order to avoid set-up or hold time violations, we very conservatively use a second pipeline stage to access the table entries. This additional pipeline stage will impose a penalty in the normal mode of operation. While we expect that the actual performance penalty would be far less than a pipeline stage (e.g., if BHT access latency is not the determining factor in pipeline stage latency), we choose this pessimistic design point as a lower bound on SRAS's benefit. In Section 2.4.1, we run experiments to assess the impact of this additional pipeline stage on the execution time in the absence of hard faults.

2.2.4 SRAS-CR Costs

The cost of a fault tolerance scheme has three aspects: hardware (area) overhead, performance (timing) overhead, and power consumption overhead. For aggressive microprocessor architectures, the performance overhead during fault-free execution is often the most critical parameter.

In order to keep the performance overhead at a minimum, buffers and tables are handled differently in SRAS. The distinct nature of buffers that makes all of their rows have equal functionality enables a no-timing-overhead implementation. Tables, however, require a definitive logical address for the data, which results in a need for an additional level of indirection. This indirection results in two gate delays in access times (e.g., for the Pentium4, an inverter delay is about 1-2% of the clock period [73]). Since gate delay will be larger than inverter delay, and since we cannot

know how much margin exists in an existing design, we very conservatively add a pipeline stage for access to tables. The additional pipeline stage results in increased latency and an increased number of stalls, and we evaluate its performance overhead in Section 2.4.1.

The increase in power consumption in SRAS-CR stems mostly from increased data read/write activity due to the check rows. Since the write/read activity is doubled, the dynamic power consumption in the array structures will roughly be doubled as well. If power consumption is still a concern, accesses to check rows can be reduced at the expense of increasing the fault detection latency.

Finally, the hardware overhead of SRAS-CR includes the need for (a) DIVA, (b) spare rows (including spare check rows), (c) one logic circuit for repair and check per array structure, (d) the per-row counters for diagnosing hard faults, and (e) two additional read and one additional write ports on the protected array structures to support simultaneous writing of the check row and reading of the result and check rows. DIVA is the primary cost yet, according to Weaver and Austin [80], a DIVA checker's size is less than 5% of an Alpha 21264 core. Thus, there is an engineering trade-off between availability and the area overhead incurred for spare rows.

2.2.5 Limitations of SRAS-CR

The implementation of SRAS-CR we present here does not tolerate all microprocessor faults. We divide these untolerated faults into three categories. First, SRAS-CR does not tolerate faults in its own logic, e.g., the pointer remapping logic or the fault map. These structures are far smaller than the structures they are protecting, which makes them less prone to hard faults, but they could still fail. Second, SRAS-CR does not tolerate a fault in a table sub-array if no more spare rows are available in that sub-array. This limitation does not apply to buffers except in the extreme case in

which every row of the buffer, including spares, is faulty. Third, SRAS-CR does not tolerate a fault in a sub-array (for a buffer or table) if all of the check rows for that sub-array are faulty.

All of these untolerated faults present the designer with a classic engineering trade-off: fault tolerance versus hardware cost. Future SRAS-CR implementations could develop hardened logic if the first fault model is considered important. The probabilities of the latter two categories can be decreased by designing the SRAS-CR protection to use more spare rows and more check rows.

2.2.6 SRAS-EDC: Self-Repair Design Without DIVA Backstop

In this section, we present a design for array self-repair that is independent of DIVA and that is fully integrated into the microprocessor datapath. The design attempts to minimize the amount of logic, particularly on critical paths. An illustration of our design (simplified for purposes of illustration) is shown in Figure 2-6. As we mentioned previously in Section 2.2.1, the microarchitecture is similar to that of the Intel PentiumPro in that the reorder buffer holds the results of completed but not yet committed instructions (rather than keeping them in the physical register file). The array structures we protect are the instruction buffer, instruction scheduling window, reorder buffer, load-store queue, and BHT. In the figure, unprotected instructions are fetched into the datapath, and protected data is eventually written back to the register file or data cache. The register file and data cache are highlighted to emphasize that they hold architectural state and that they cannot be recovered using the core's misprediction recovery mechanism. Note that, with minor modifications, our scheme could be adapted for use in microarchitectures with register update units (RUUs) or microarchitectures that keep the results of completed but uncommitted instructions in the physical register file and use explicit register renaming with a map table. Our design treats the combinational logic that manipulates the data that flows through the microproces-

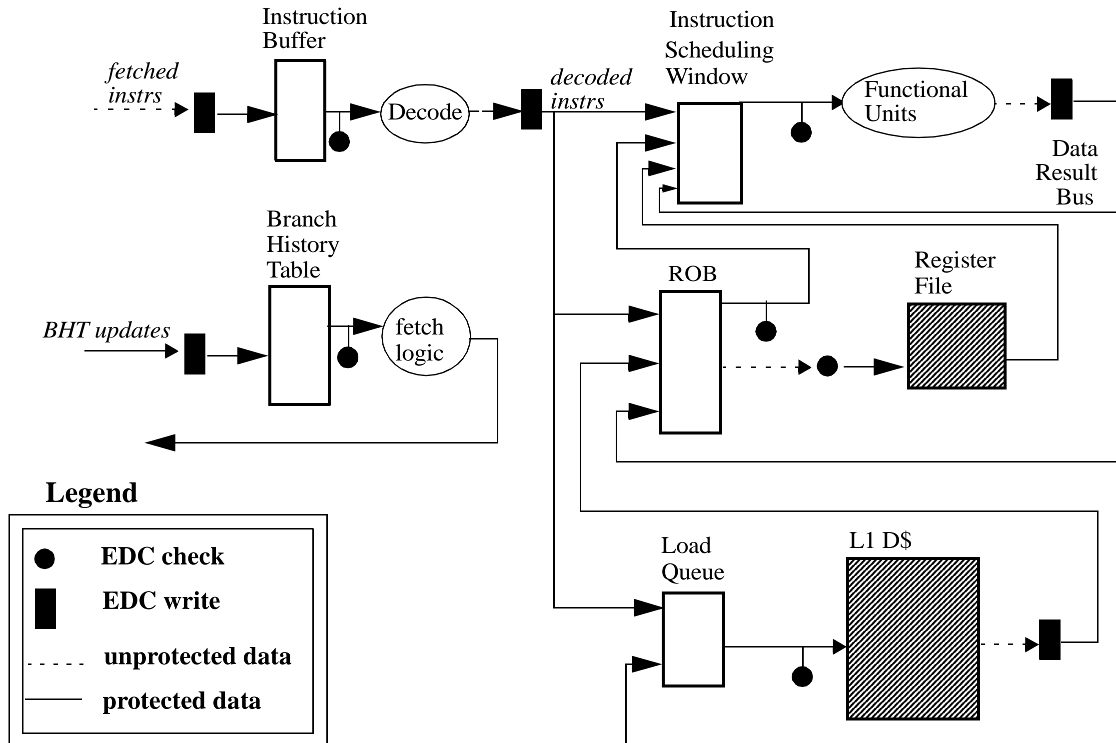


Figure 2-6. Datapath Design with SRAS-EDC

The register file and L1 data cache (L1 D$) are highlighted to emphasize that they hold architectural state. This simplified figure ignores the store queue, since stores are handled just like non-load instructions, except that they write their results to the L1 D$ instead of the register file.

sor (e.g., instruction decoders, functional units) as black boxes. Protecting this logic from hard faults is an orthogonal issue.

As an instruction progresses through the pipeline, every time it is modified, an EDC write must occur. As Figure 2-6 shows, this activity occurs after instruction fetch, instruction decode, ALU operation, memory reads, and before updating the BHT, assuming it is optionally protected. With the exception of the BHT update, these EDC write operations must be on the critical path of the pipeline, and thus add additional latency to the instruction's processing time. The use of EDC, as opposed to ECC, is advantageous in that it provides for a lower-latency operation that takes less logic to implement in the timing and space-constrained pipeline. EDC must be checked after any access to a datum contained in a protected structure. However, the only time that this EDC check

activity is on the critical path of an instruction's execution is when the instruction's result is to be committed to architectural state at the end of the pipeline. At all other times, the datum can be used by a subsequent pipeline stage without knowing the EDC result, since the later discovery of an error in the EDC check can be contained by flushing the pipeline.

2.2.6.1 Detection and Diagnosis

SRAS-EDC uses error detecting codes (EDC) to detect and diagnose errors in array rows. There are numerous kinds of EDCs, including parity and cyclic redundancy check (CRC) codes. EDCs add some number of check bits, *k,* to the original *d* data bits, and the tradeoff is between the cost due to the number of check bits added and the added error detection capabilities of having more check bits. For example, a single parity bit adds a *1/d* cost and can detect all single-bit errors. For implementation purposes, we prefer a *separable* EDC, i.e., the check bits are not interleaved with the data bits. Thus, each array row consists of *d* data bits followed by *k* check bits. We also want an EDC that can detect all single-bit errors and many types of multiple-bit errors, particularly unidirectional errors (i.e., all 0->1 or 1->0). Many EDC options exist—a designer can choose the EDC that best suits the system, based on the tradeoff between error detection capability and implementation cost. Because of our fault model, we choose Berger codes [10] to protect all arrays except the BHT, since Berger codes can detect all single-bit errors and all unidirectional multiple-bit errors. A Berger code will detect all single stuck-at faults and coupling faults (from one bit to any number of neighboring bits). In a Berger code, the *k* check bits are the binary representation of the number of zeros in the original data, and thus $k = \lceil \log_2(d+1) \rceil$. For the BHT, which has only 2-bit entries, we simply use a parity bit for EDC.

As in SRAS-CR, to distinguish hard faults from soft faults, we add a small counter to each row that is incremented for every error detected in it, and all counters are periodically cleared. If an

error increments a counter such that it exceeds a specified threshold, then this row is considered to have a permanent fault; otherwise the error is considered transient. All data written into arrays is protected with EDC, and all data read from arrays has its EDC checked. We also maintain EDC bits in the register file, in order to not have to re-compute EDC for data that is read from the register file to be written into the instruction window. Nevertheless, we are not protecting the register file from hard faults—a hard fault would be detectable but unrecoverable.

The only six instances in which EDC logic (writing EDC bits to the end of a datum or checking EDC bits) can potentially impact performance are when:

- Fetched instructions go through logic that adds EDC bits to them before inserting them into the instruction buffer.
- Decoded instructions go through logic that adds EDC bits to them before inserting them into the instruction scheduling window. EDC needs to be recomputed here, since the process of decoding the instructions modifies their data payload.
- Data produced by functional units goes through logic that adds EDC bits before being written into the instruction window (as operands) and the ROB (as results). EDC needs to be recomputed here, since the functional units produce new data. This EDC logic could be associated with the functional units or with the data result bus. An optimization is to compute the EDC (for the outputs of the functional units) in parallel with the outputs. This requires more hardware but hides the latency of the EDC logic, and we will explore the potential of this approach in our evaluation in Section 2.4.1. In this work, we do not consider use of *self-checking circuits* [79], in which EDC codewords (using *arithmetic codes*) are produced by the combinational logic (e.g., functional units). This technique would enable us to check the functional units themselves (but not to map out hard faults in them), and it would also remove the need

for this EDC recomputation logic. The techniques we use and results we present in our evaluation represent a pessimistic performance bound since a self-checking implementation would remove all EDC generation logic from the critical path of the processor pipeline.

- Data loaded from the L1 data cache goes through logic that adds EDC bits before being written into the ROB. EDC needs to be recomputed here, since we do not assume that the caches implement the same EDC. If we were to relax this assumption, then this logic for recomputing EDC would no longer be necessary.

- Data from the ROB goes through logic that checks the EDC before being committed into the register file (or into the L1 data cache, for stores). In the figure (which omits stores, for clarity), this data is shown as unprotected (before it is checked) despite coming from the protected ROB. This is because, unlike for other structures, the EDC check on this data cannot be done later and thus undo the effects of writing this potentially erroneous data into the register file or data cache.

- Updates to the branch history table go through logic that adds a parity bit. However, checking the parity bit of data that is read from BHT is off the critical path.

In the first five of these situations, EDC logic is on the critical path, and we pessimistically assume that we must add an extra pipeline stage to accommodate this latency. The exception is adding the parity bit to the BHT—we assume that this simple operation will not force the addition of a pipeline stage. In all other instances, EDC logic is *off the critical path*. For example, when instructions pass from the instruction buffer to the instruction window (after being decoded and renamed), their EDCs are checked off the critical path. That is, erroneous data could be written into the instruction window before the EDC check is complete; however, the EDC check will fail soon thereafter and trigger a system recovery which will eliminate the effects of the error before

they can be committed to architectural state. Other EDC checks are between the instruction window and the functional units, between the load queue and the data cache, and between the ROB and the instruction window.

One potential challenge for fast implementation of EDC (or ECC, for that matter) is that partial writes to a structure (i.e., writes that do not modify the entire data) turn into RMW operations. Recall that this limitation is also present for SRAS-CR for any write that will have a check performed. The read is necessary to help compute the EDC over the entire data before writing it. Since RMWs are slower and require extra array bandwidth, we would like to avoid them if possible. Our solution is to compute EDCs over independently written fields of array rows, instead of over the entire row, in order to avoid any possible partial writes. For example, in the ROB, we compute separate EDCs for the result data and for the rest of the entry. Thus, when the entry is allocated, we must compute both, but this is no more complex than computing it over the whole entry. The key savings is when the result is written during instruction completion, since we no longer need to do a RMW.

2.2.6.2 Recovery

Recovery is implemented with the microprocessor's normal misprediction recovery mechanism. Thus, unlike SRAS-CR, SRAS-EDC does not need DIVA. This recovery mechanism effectively deletes all speculative, uncommitted microprocessor state (e.g., contents of the instruction buffer, instruction window, ROB, etc.), but it cannot undo changes made to architectural state such as the register file. This is why the EDC check between the ROB and register file is on the critical path.

2.2.6.3 Remapping

We use the same techniques as SRAS-CR for mapping out faulty rows of arrays.

2.2.6.4 SRAS-EDC Costs

The costs for SRAS-EDC are less than those of SRAS-CR in two important ways. First, SRAS-EDC does not require DIVA. Second, SRAS-EDC does not require all of the extra reads and writes that were necessary for the check rows. However, SRAS-EDC does add some hardware for performing EDC computations. It also adds some performance overhead because of those instances in which EDC logic is on the critical path.

2.2.6.5 Limitations of SRAS-EDC

There are a few limitations of SRAS-EDC. First, the fault coverage is limited by the strength of the particular EDC that we choose. This is parameter can be tuned to allow a designer to trade off error detection capability against implementation cost. Second, we can only protect structures that do not hold committed architectural state. Thus, we can protect the ROB, LSQ, IQ, IW, etc., but we cannot protect the register file, processor status word, condition codes, etc. In order to extend SRAS-EDC to cover this portion of the microprocessor core, an additional state save and recovery mechanism would be required (e.g. a checkpointing scheme).

2.2.7 Applicability of SRAS to Specific Structures

In developing SRAS-CR and SRAS-EDC, we have studied the common structures within the microprocessor core and applied SRAS techniques to those structures that we believe support it economically (that is, without undue redesign or timing constraints). While we generalize structures as buffer-like or table-like, each structure must be considered in detail to understand how SRAS can be made to work on it. This section presents the detailed assumptions about the different structures we studied to give the reader better intuition in applying SRAS techniques to a specific design that we have not specifically addressed with this study.

2.2.7.1 Instruction Buffer

The instruction buffer is a straight-forward structure to protect with SRAS. As the holding place for fetched instructions awaiting decode, this buffer is a FIFO queue, with each entry written once. There is no requirement to modify its basic structure to accommodate SRAS application.

2.2.7.2 Instruction Scheduling Window

After instructions are decoded, they are cached in this structure until their operands are ready and functional units are available to execute them. Allocation of entries is buffer-like in nature and we treat it thusly for SRAS application. In order to do this, however, we must consider the following aspects of the instruction scheduling window that are not queue-like. First, the structure is implemented as a content-addressable memory (CAM) to enable wake-up and select logic to properly find ready instructions as well as to allow operand readiness to be properly updated each cycle. Second, this structure is the beginning of the out-of-order execution of the microprocessor core. Instructions are removed as they become ready, not in FIFO order. Typical implementations perform a compaction of the structure at the end of each cycle to keep oldest instructions near the head of the queue and to simplify allocation of entries in the next decode cycle. Finally, the contents of this structure will typically be updated between the initial write to it and its eventual use and entry retirement.

For SRAS-CR, these factors are mitigated by performing a full write of the entry at its point of allocation and performing the check at that time. As mentioned in the discussion of SRAS-CR, pseudo-random data from the tick counter is used to populate the uninitialized fields of this structure to allow for the check to be calculated properly without requiring partial updates be converted to RMW activities and to have a fixed upper bound on the number of check circuits required to perform checks (for n-wide decode circuitry, we need n check circuits). Subsequent overwriting of partial data and movement during compaction is effectively ignored by SRAS-CR. This is tolera-

ble since DIVA will correct any errors introduced by moving a good datum to an array entry that has a hard fault present that has yet to be deconfigured.

For SRAS-EDC, EDC must be recalculated for every update to the structure. We can avoid RMW requirements by dividing the EDC into separate EDC fields for each of the written sub-pieces of the entry. This also provides the advantage of only requiring $2n$ EDC calculation circuits, since at most n operands will become ready in a given cycle in an n-wide processor. The calculated EDC for a particular operand becoming ready is independent of the rest of the instruction window data. This allows a single EDC calculation to be written multiple times at all applicable locations in the instruction window. So, in a given cycle, n EDC calculations will be required for the incoming n decoded instructions and the newly-computed n ready operands (making for a total of $2n$). Compaction activity is not a problem, since the EDC travels with the entry and remains valid during the compaction (that is, compaction performs no update on the data, only moving it to a new location in the buffer). As with SRAS-CR, the number of EDC checking circuits required for the structure is equivalent to the issue width of the processor.

2.2.7.3 Load-Store Queue

The LSQ is FIFO in nature, but is also implemented as a CAM to allow for searching. This additional implementation complexity does not adversely impact either of the SRAS schemes. SRAS-CR again uses pseudo-random data if necessary to perform the check at the time of entry allocation. SRAS-EDC must maintain $2n$ EDC calculation circuitry sets (one for the initial write of the entry and one for address calculation arrival from an ALU) in order to allow for a peak sustained memory bandwidth of n instructions per cycle on an n-wide processor. Only n copies of the EDC check circuit are required. As with the instruction scheduling window, the sub-fields of an

entry may have their EDC calculated separately to simplify the EDC calculation circutry's implementation.

2.2.7.4 Branch History Table

The BHT is a table with addressable content on a very small granularity. The descriptions of SRAS operation for table structures were crafted with the BHT as a motivating example. For other tabular structures, the aforementioned techniques of writing pseudo-random data (for SRAS-CR) or splitting the table entry into separate EDC fields (for SRAS-EDC) may be applicable.

2.2.7.5 Reorder Buffer

The ROB is a FIFO queue with the potential of multiple partial writes during the lifetime of an instruction. Issues here are similar to those found in the instruction scheduling window. The same techniques would apply here for the two SRAS methods.

2.3 Online Diagnosis of Hard Faults in Microprocessors

With a fine-grained hard-fault tolerance mechanism for array structures established, we now seek to extend our fine-grained techniques to include capability to diagnose hard faults in the microprocessor core. By using existing techniques, including the SRAS methods developed above, we will arrive at a diagnosis mechanism capable of detecting hard faults in a bounding box that surrounds a majority of the microprocessor core logic from decode through retirement of instructions. As with SRAS-CR, we use DIVA[6] as an error detection and correction mechanism upon which we develop our diagnosis technique. We begin with a presentation of existing diagnosis alternatives in Section 2.3.1 before we describe our diagnosis mechanism in detail in Section 2.3.2. This section concludes with a presentation of deconfiguration techniques in Section 2.3.3 and a discussion of the limitations of our presented method in Section 2.3.4.

2.3.1 Fault Diagnosis

DIVA checkers do not provide fault diagnosis. They are only capable of detecting and correcting errors, not determining their underlying causes. For transient faults, this is appropriate, since the desired remedy never involves altering the configuration of the core. For hard faults, however, we show in Section 2.4 that it is often desirable to deconfigure part of the superscalar core in order to prevent frequent errors and the performance penalty that frequent pipeline flushes from DIVA corrections (or redundant thread corrections) would require.

We define sub-structures within the processor core that we wish to be able to deconfigure as field deconfigurable units (FDUs). To diagnose hard faults in the processor core, we first have to select the FDU granularity at which we wish to be able to diagnose. Many structures are replicated within a typical superscalar core, and the granularity of replication represents a natural FDU granularity. The choice of FDU is a design decision for a given implementation. Because deconfiguration is more easily achieved with this FDU selection, we favor it over an FDU selection that seeks to have equal amounts of logic in each FDU. For the processors that we model in our evaluation, the identified FDUs for which we track diagnosis information are: individual entries in the instruction fetch queue (IFQ), individual reservation stations (RS), individual entries in the load-store queue (LSQ), individual entries in the re-order buffer (ROB), individual arithmetic logic units (ALU), and the individual DIVA checkers. While our chosen processor designs have only one of some of the more complex ALUs (for example, the integer multiplier), we include them in our diagnosis evaluation to show that the diagnosis is capable of identifying hard faults in these units. We have chosen a fairly fine FDU granularity, but one could choose coarser or even finer granularities if so desired; we discuss this engineering tradeoff later. The hardware bounds of our diagnosis mechanism are the components in which the selected error checker (in our design, DIVA) can

detect a fault. Therefore, we do not consider the register file, because DIVA cannot recover from errors in it.

2.3.2 A New Online Diagnosis Mechanism

We propose to dynamically attribute errors to FDUs as the system is running. Given an error detection mechanism, if an instruction (or micro-op, in the case of IA-32) is determined to be in error, the system records which FDUs that instruction used during its lifetime. If, over a period of time, more than a pre-specified threshold of errors has been attributed to a given FDU, it is very likely that this resource has a hard fault.

To track each instruction's FDU usage, bits are carried with each instruction from the point of FDU usage to commit. For those structures that the instruction owns at commit, this information is already implicitly available and no extra wires are needed to carry this resource usage info through the pipeline. In our modeled processor, the ROB entries and DIVA checkers use implicit tracking. For the remaining FDUs, the number of bits required is a function of the size of the structure and the granularity into which we are allowing it to be sub-divided for later deconfiguration. This represents an engineering trade-off in our design that will allow implementations to select the appropriate FDU granularity/overhead trade-off. For typical superscalar microprocessor designs, including those that we evaluate in Section 2.4, roughly 20 bits are required to track this fine-grained FDU utilization information. Carrying these extra bits through the pipeline incurs two costs: pipeline latches will be marginally wider and there will be more wires to route through the pipeline. However, compared to the 64-bit operands that are carried through the pipeline, these extra bits are a small addition, especially since not all of the bits need to traverse the whole pipeline.

For each FDU we track, the processor maintains a small, saturating error counter. The purpose of the error counter is to differentiate hard faults from soft faults. At the scope of the error detection and correction mechanisms considered (that is, at the instruction granularity), hard faults are not distinguishable from soft faults at the time an error is detected and corrected. For hard faults affecting frequently used structures, we observe an error detection and correction rate that is orders of magnitude higher than that observed for transient faults. Occasional corrections due to soft faults do not trigger diagnosis because they do not saturate the error counter for any given FDU in the system. Periodic clearing of the error counters prevents soft fault corrections from accumulating to a point where diagnosis is triggered.

2.3.2.1 Design Issues

Using saturating error counters for diagnosis of hard faults presents four challenges. First, after the FDUs have been selected and configured for diagnosis in an implementation of our mechanism, all remaining logic for which the error detection and correction mechanism detects and corrects errors must also be tracked by our diagnosis scheme. For our design, this critical logic includes all logic that is not within an FDU but that is in the portion of the superscalar core for which DIVA is capable of detecting errors. This includes instruction issue, any singleton arithmetic logic units (ALUs) (for example, a floating point multiply/divide unit), floating point ALUs, and any common datapaths that all instructions must traverse.

The second issue with using saturating error counters is that transient errors must not lead to above-threshold error rates. Thus, we must have error counter thresholds that are not too small, and the microprocessor must periodically clear the error counters to prevent transient errors from accumulating past the hard fault threshold. The frequency of counter clearing is an adjustable parameter that depends on expected transient error rates. Counter clearing is a low-cost operation,

so we recommend clearing once every ten seconds, even though current terrestrial transient fault rates do not approach this frequency. This rate is based upon our experimental results for latency to diagnose hard faults. Our experimental results show that the latency to diagnose a hard fault in the FDUs we evaluate is less than 1/10th of a second at multi-gigahertz frequencies, even in infrequently-used FDUs. By clearing at an interval well above the diagnosis latency of FDUs we care to diagnose, we ensure that we will diagnose hard faults that greatly affect system performance if they are allowed to continue to cause error detection and correction to occur. If diagnosis spans a clearing interval, we are merely postponing the deconfiguration temporarily. Also, if a hard fault is detected and deconfiguration is activated, the deconfiguration process clears the error counters.

Third, the error rate threshold for a resource must be related to its usage. For example, a very high threshold for a resource that is rarely used will preclude the system from ever diagnosing a hard fault in it. To illustrate this, consider the case where we have a single adder and two ROB entries. Assuming we use the adder and one of the ROB entries each cycle, we can observe that a fault in the ALU will cause both ROB entries' error counters to accumulate errors at a rate of 1/2 that of the adder. To avoid mis-diagnosis, we would need the adder's saturation value to be greater than that of an ROB entry, but not more than twice the ROB entry value. Thus, for frequently utilized FDUs, a larger counter value is required to prevent the mis-diagnosis of a fault in an upstream or downstream structure.

The final challenge is that the chosen FDUs must be used reasonably independently. Otherwise, for example, if every time an instruction uses FDU A it also uses FDU B, then the diagnosis mechanism will not be able to distinguish between a hard fault in A and a hard fault in B. To guarantee that instructions take many different and independent paths through the pipeline, we slightly change the scheduling of resources that are normally scheduled non-uniformly (e.g., higher prior-

ity for ALU_0) to add a round-robin aspect to it. For example, instead of always allocating the lowest-numbered ALU that is available, the microprocessor allocates available ALUs in a round-robin fashion. Otherwise, the usage of ALU_0 could be significantly greater than that of other ALUs and thus preclude hard faults in them from being diagnosed (since the thresholds assume uniform utilization). This scheduling modification is not necessary for resources that are naturally scheduled uniformly, like ROB entries. We found that round-robin scheduling alone does not avoid all lockstep allocation of resources, though. For example, with three ALUs and three DIVA checkers, we found that a long string of instructions that all used ALUs led to undiagnosable errors. In one particular scenario, an instruction that used ALU_0 always used $Checker_1$, ALU_1 was perfectly correlated with $Checker_2$, and ALU_2 was perfectly correlated with $Checker_0$. To avoid this lockstep allocation, we introduced a small amount of pseudo-randomness into the scheduling of checkers. Every cycle, the first checker to be considered for allocation is determined based on pseudo-random data (e.g., low order bits of the tick counter), and then subsequent checkers are allocated sequentially (mod width) after the first one. This pseudo-randomness, combined with round-robin scheduling, prevents lockstep allocation and achieves reasonably uniform utilization of each set of identical FDUs.

2.3.2.2 Heuristics for Choosing Error Counter Values

Given these four challenges, we developed a heuristic for choosing appropriate threshold values for the saturating error counters. As it is always possible to craft an instruction sequence that leads to saturation of the wrong counter, the best that we can do is to choose saturating values and then verify correct diagnosis operation via simulation. Using this heuristic for the designs we evaluate in Section 2.4, we will see that it does provide effective threshold values that lead to low-latency diagnoses of a wide range of FDUs. The heuristic is as follows:

4) Select a minimum power-of-two threshold value well above what transient or intermittent faults would cause in a counter-clearing interval.

5) Segregate FDU types by the population of units for each type. For FDUs that have a population that is not a power of two, round the population to either the next larger power of two, if it is a heavily-utilized resource, or the next smaller if it is a less-heavily utilized resource. Resource utilization information may need to be gathered via simulation of representative workloads at this point. Group like-population FDUs together. Assuming that there is some logic for which error detection and correction can contain a fault, but for which there is no associated FDU, create a singleton group for "critical logic."

6) Assign the minimum threshold chosen in step 1 to the highest-populated FDU group.

7) Assign the next power-of-two as the error counter threshold for the next-most-populated FDU group.

8) Repeat step 4 for all remaining FDU groups, assigning the highest threshold to the "critical logic" group.

9) Simulate the processor with a representative set of workloads and FDU faults to verify that the thresholds chosen cause the diagnosis mechanism to converge on the faulty FDU.

10) Using the simulation results from step 6, reduce the threshold by a factor of two (one bit) for those items whose diagnosis latency is large. If this threshold reduction results in no FDUs with an error counter threshold in the middle of the threshold range, reduce all higher error counter thresholds by a factor of two. This will result in a set of error counters whose bit width is monotonically increasing, without any gaps from lowest to highest. Repeat the simulation to verify correct operation.

Table 2-2. Error Counter Thresholds

FDU	threshold	storage requirements for diagnosis
instruction fetch queue entry	32	5 bits/entry
reservation station	32	5 bits/entry
reorder buffer entry	16	4 bits/entry
load/store queue entry	16	4 bits/entry
integer ALU	64	6 bits/unit
floating point ALU	64	6 bits/unit
integer multiplier	32	5 bits/unit
floating point multiplier	32	5 bits/unit
DIVA checker	64	6 bits/checker
critical logic (issue, etc.)	128	7 bits

In Table 2-2, we list the counter thresholds for the FDUs we consider in this paper, including the per-unit storage cost for each FDU's counter. These values were derived for our three evaluated processor design points using the above heuristic with a minimum threshold value of 16. For resources that are less utilized, such as the floating point units, our mechanism may take additional time to diagnose, even with the lower threshold than their more heavily-utilized integer counterparts. Any hard fault that gets exercised so rarely as to not exceed our error counter threshold between periodic counter zeroing is also so rare that it incurs little performance penalty for its infrequent error recoveries. In this situation, simply using DIVA to correct errors due to a hard fault in a lightly-utilized FDU is sufficient. The key observation is that our scheme can diagnose hard faults in the highly utilized resources, so that the microprocessor avoids frequent recoveries.

2.3.2.3 Discussion

We include the DIVA checkers in the error diagnosis design, so that we can enable the microprocessor to tolerate hard faults in the checkers. Since a *k*-way superscalar microprocessor requires approximately *k* checkers to avoid having the checkers become a bottleneck, we would like to be able to tolerate a hard fault in one of them by leveraging their redundancy.

Using DIVA for error detection and correction provides three unique issues related to diagnosis and deconfiguration of a hard-faulted unit. First, uncached loads and stores commit without any redundant check of the operation, making them undiagnosable. A fault affecting the logic unique to these operations will not be covered by our mechanism. The system will perform exactly as it would if it only had DIVA checkers active. Second, the microprocessor is vulnerable to transient errors in DIVA checkers, but DIVA assumes that small checkers can be designed to be more resilient to transient faults by using more robust feature sizes. Third, because the microprocessor trusts a DIVA checker until its error counter exceeds its threshold, the microprocessor is vulnerable to incorrect execution in the window between when a hard fault occurs in a checker and when it diagnoses that the checker is the culprit. We further discuss this window of vulnerability in Section 2.3.4.2.

There are certain scenarios in which the diagnosis mechanism can temporarily deconfigure a fault-free FDU. A transient or hard fault in our added hardware—error counters, wires for tracking resource usage, and deconfiguration logic—could lead to deconfiguring a fault-free component. Also, the use of saturating counters for the FDUs within the processor introduces the possibility that the wrong unit's counter will saturate first for a particular instruction sequence. To address this issue, we use an iterative diagnosis process. Diagnosis is not considered complete until fault rates fall below a hard-wired threshold set by the designer We set this threshold sufficiently high to allow for all hard faults that we wish to diagnose to be accounted for. The final unit deconfigured before this error rate change is considered faulty, while all other units deconfigured in the affiliated diagnosis cycles are returned to operation. In general, if deconfiguration does not help (i.e., as unit(s) are deconfigured, error counters continue to saturate in close temporal proximity), then the system can reconfigure the previously mapped out unit(s) back into the system (under the common

assumption of one hard fault at a time) once the correct unit has been identified and deconfigured. Our evaluation in Section 2.4 will show that one iteration is sufficient a vast majority of the time.

The microprocessor also tolerates faults in the error counters by testing them. After clearing the counters, it checks that they are indeed all zero. It also uses a small amount of hardware to periodically test that the counters can be incremented correctly. If a counter is faulty, the corresponding FDU is then permanently either configured or deconfigured, based upon whether it is mapped back in or left deconfigured. Mapping it back in leaves the system vulnerable to a hard fault in this FDU, but leaving it deconfigured is potentially a loss of useful hardware.

2.3.2.4 Alternative Design Options

There exist other ways to perform fault diagnosis. The most obvious approach is to use TMR—if two modules produce one result and the third module produces a different result, then the system diagnoses the third module as faulty (assuming a single-fault model). TMR, however, has a 200% hardware and power overhead.

Another well-known diagnosis approach is built-in self-test (BIST). After detecting an error and determining that it is due to a hard fault (e.g., by detecting it repeatedly), systems with dedicated BIST hardware can test themselves in order to diagnose the location of the hard fault. To its advantage, unlike our new diagnosis mechanism, BIST does not have to worry about the statistical nature of online error counting. BIST can be applied to a microprocessor like the ones we study, and one concurrent BIST mechanism can be used for all components in the path, although the number of BIST test vectors to generate—either deterministically or pseudo-randomly—would be extremely large. BIST requires the processor to be offline for testing to occur. Our online error counting differs from BIST by diagnosing faults via the observation of the execution of actual software with the software's instructions acting as test vectors and the error detection and correction

acting as output verifier. This ensures that we always have a test vector that exposes a detected fault. Finally, BIST adds performance overhead due to the extra multiplexers that choose between normal inputs and BIST inputs. Unlike our diagnosis overhead, this overhead is on the critical path of instruction flow through the processor. Since many processors have some form of BIST support already in their design, use of our mechanism presents an opportunity to remove this hardware from the critical path, replacing BIST with our mechanism.

Within our diagnosis mechanism, there are also design options. If, instead of using DIVA, we used redundant threading for error detection and correction, this would also affect our diagnosis mechanism. DIVA assumes that the checker core is always fault-free and thus it can diagnose with only two copies of a given unit (e.g., the multiplier in the out-of-order core and the multiplier in the checker). If a redundant threading scheme is used for detection and correction of hard faults, it must use independent resources for each of the primary and redundant threads in order to guarantee that results are not derived from the same faulty hardware. Since with redundant threading, there is no known-good unit, we need at least three copies of a given unit to ensure forward progress is achieved in the presence of a hard fault. Otherwise, for example, a hard fault in one of two multipliers would cause repeated miss-matched results with no way to determine which result is correct. In this case, the instruction would replay continually until a higher-level deadlock detection mechanism activated. With at least three copies of a unit, the two fault-free copies will calculate the correct result, allowing us to isolate the faulty functional unit and then increment its associated error counter.

Finally, an alternative, related diagnosis mechanism bears mentioning. As an alternative to keeping saturating error counters for each FDU and all logic covered by the chosen error detection and correction mechanism, a microarchitect could opt to have a single, saturating error counter

that triggers diagnosis. This counter, when saturated, would lead the system to replay the faulted instruction, deconfiguring and replacing each FDU involved in the last erroneous result until a correct result is obtained. At that point, the currently-deconfigured FDU would be deemed faulty and would remain deconfigured from the system, with normal operation resuming. This method presents three drawbacks. First, to use this alternative, the microarchitecture would have to support directed steering of instructions through specific FDUs to allow for multiple replays with only a single suspect removed from the processing of each replay. This would add additional complexity to every stage of the pipeline. Second, if a transient fault happens to cause the diagnosis in this alternative scheme, diagnosis will take the maximum amount of time and will result in no unit deconfigured, requiring a subsequent diagnosis attempt on the next encountered error. Finally, if a transient occurs during diagnostic replay, it will result in either the diagnosis missing the suspect unit, requiring another round of diagnosis, or a double-fault case, which greatly complicates error detection. Given these issues, we chose the use of error counters for each FDU, which leads to a single deconfiguration action upon saturation without requiring any directed replay of instructions.

2.3.3 Deconfiguring Faulty Components

After an FDU has been diagnosed as having a hard fault present, deconfiguring the faulty FDU is desired to avoid the frequent pipeline flushes that DIVA would trigger due to continued manifestation of the fault. In this section, we describe several pre-existing methods for deconfiguring typical microprocessor structures, plus a new way to deconfigure a faulty DIVA checker.

For circular access array structures—such as the instruction fetch queue (IFQ), reorder buffer (ROB), and load/store queue (LSQ)—previous work has shown how to add a level of indirection to allow for de-configuration of a single entry with little additional latency added to access time for the structure [64]. If we use SRAS-style remapping, each structure maintains a fault map. This

fault map information feeds into the head and tail pointer advancement logic, causing the advancement logic to skip an entry that is marked as faulty. If cold spares are available, as in our SRAS designs, shown in Figure 2-4, the structure size can be maintained at the original processor design point. If no spares are provisioned, which is what we assume in our analysis, then the structure size must be updated when the fault map is updated.

For some tabular (i.e., directly addressed) structures—such as reservation stations, register files, etc.—a simple solution is to permanently mark the resource as in-use, thus removing it from further operation [64]. Once again, use of SRAS assumes that cold spares may be available, as shown previously in Figure 2-5, even though we assume no provisioning of cold spares in the evaluation of our new diagnosis mechanism.

For a functional unit (ALU, etc.), similar to a reservation station, we can mark the resource as permanently busy, preventing further instructions from issuing to it [64]. Cold sparing of functional units is possible, but it may require too much hardware area, as functional units are relatively large compared to individual ROB entries or reservation stations. We focus on using existing redundancy, since the cost of adding extra redundancy may be too great for commodity microprocessors.

For one of the multiple DIVA checkers, we can map it out if we diagnose it as being permanently faulty. Depending on how DIVA checkers are scheduled, deconfiguration is just as simple as for ALUs; just marking a faulty checker as permanently busy will deconfigure it. Prior work has not looked into deconfiguring DIVA checkers, because no fault diagnosis schemes prior to this work could diagnose hard faults in a checker.

2.3.4 Costs and Limitations

The design that we have presented in Sections 2.3.1-2.3.3 is not free, nor is it without limitations. In this section, we present its hardware costs and limitations.

2.3.4.1 Hardware Costs

We add hardware to an unprotected microprocessor to achieve hard fault tolerance. The largest, single addition to the processor is the DIVA checkers, each of which has been estimated at 6% of the size of an Alpha 21264 core [80]. In addition to DIVA, which provides benefits even without our additions, we also add: error counters, wires for tracking each instruction's resource usage, and logic for deconfiguring FDUs. None of these additional hardware costs are large; moreover, they can all be reduced at the expense of a coarser granularity of diagnosis and deconfiguration. For example, we can share one error counter and one wire among k entries in the instruction window, at the cost of having to deconfigure all k entries if any of them incurs a hard fault.

2.3.4.2 Limitations

We now discuss three limitations of our current implementation and approaches for addressing them in the future. First, there are certain structures that we either cannot protect or that are very difficult to protect. Our current implementation cannot protect the register file, because it is part of the recovery point for DIVA recovery. We cannot diagnose faults in singleton resources that are used with a majority of instructions, due to ambiguity reasons stated at the end of Section 2.3.2. Examples of these resources include issue logic and common datapath lines. These singletons are always in lock-step scheduling with each other. Utilizing a modular implementation for these currently monolithic structures could make them configurable as FDUs in our diagnosis scheme, but such designs are beyond the scope of this work.

Related to this issue is the impact of hard faults in the datapaths and unique logic for each FDU. For some FDUs selected, there is a unique set of logic and data paths that will affect correct execution for a subset of instruction paths through the processor if hard faults are present, but for which diagnosis will lead to deconfiguration of a downstream unit. In these instances, the deconfiguration action results in discontinued use of the faulted portion of the circuit via deconfiguration of the downstream FDU, so the right thing happens with our diagnosis mechanism despite the problem actually residing in a different FDU.

For example, consider bypass paths between ALUs. A fault in a bypass path will be flagged as a fault in the destination ALU by our mechanism, even though that ALU is able to correctly process instructions where the bypass path is not active. By discontinuing use of the ALU, however, we observe that the bypass path is no longer used, thus eliminating the fault from further activation. To prevent this effect, we could treat bypass paths as separate FDUs, but their deconfiguration would not be straightforward, so we choose to lump them with the ALU FDUs for simplicity of the overall design. The tradeoff here is that a fully-functional ALU is deconfigured to prevent the effects of a hard fault in a bypass path.

Second, there is a window of vulnerability in which a faulty microprocessor can unwittingly produce erroneous results. Being able to deconfigure a faulty DIVA checker enables the microprocessor to improve reliability by preventing the fault from continuing to silently corrupt system state; in a DIVA-only system, it would go unnoticed until visible data corruption was recognized by a downstream entity. However, there is still a window of vulnerability between when the hard fault occurs in the checker and when it is diagnosed and deconfigured. In that window, a number of instructions equal to the error counter threshold for the checker times the number of DIVA checkers could have been committed in error, since DIVA checkers assume they are correct in the

case of a mis-comparison. Without a higher-level recovery scheme, such as checkpointing, this erroneously committed state represents an unrecoverable error. It should be noted that DIVA also can cause silent data corruption when a transient fault affects a checker. Since this is not detectable by DIVA or our diagnosis mechanism, it remains an exposure of any DIVA-based system.

Finally, because we elected to use DIVA in our designs, we are unable to detect and correct problems in uncached loads and stores. This is a limitation of DIVA that we inherit. This adds complexity to recovery, particularly in the case where the checker is at fault. Discussion of techniques to work around this limitation is beyond the scope of this work. This problem is not new to checkpointing research. If a designer requires containment of this escape in the scheme, an appropriate checkpointing scheme will be required. The use of an alternative error detection and correction mechanism, capable of detecting and correcting these errors, would also correct this issue.

2.4 Evaluation

Having presented SRAS and SRAS-EDC as techniques for deconfiguration of faulty array substructures and then developing a fine-grained fault diagnosis mechanism for the microprocessor core, we now present our evaluation of these designs. First, we present the common experimental methodology that was used in this work in Section 2.4.1. Our evaluation starts with SRAS in Section 2.4.2 and then concludes with analysis of our diagnosis mechanism in Section 2.4.3.

2.4.1 Experimental Methodology and System Model

To evaluate our designs' operation under the fault models considered, we modified sim-mase, as made available by SimpleScalar [5]. For SRAS, we model a dynamically scheduled microprocessor that is similar to currently available single-threaded microprocessors, such as the Intel Pentium4 [25] and Alpha 21364 [24]. The details of the target system are shown in Table 2-3. We

Table 2-3. SRAS Target System Parameters

pipeline depth	22
pipeline width	3
instruction fetch buffer	64
scheduling window	32
load-store queue	48
reorder buffer	126
functional units	4 integer adders and multiplier, 1 FP adder, 1 FP multiplier
branch predictor	gshare: BHT is 4096 entries, BHT entry is 2-bit counter, BHR is 8 bits
registers	192
L1 D-cache	8K total size, 4-way, 2-cycle
L1 I-cache	8K total size, 4-way, 2-cycle
L2 cache	256K size, 8-way, 7-cycle

protect the instruction buffer, instruction scheduling window, reorder buffer, and load-store queue with SRAS techniques.

For the evaluation of our online diagnosis mechanism, we model three separate microprocessor designs, each patterned after an existing SMT-enabled, commodity microprocessor design. The first design, *Narrow*, is a superscalar processor that is patterned roughly after the original, pre-SMT-enabled Intel Pentium 4 [14, 25]. We modify this design to include like-sized caches to the SMT-enabled design points to avoid performance side-effects from mis-matched supporting structures. The second design, *Deep-Narrow*, is a more deeply-pipelined implementation of Narrow, patterned on current SMT-enabled Intel Pentium 4 designs [14]. Deep-Narrow differs from Narrow in the depth of its pipeline, carrying an additional 11 stages to allow for faster clocking. The final processor configuration, *Short-Wide*, is inspired by the AMD Athlon/Opteron processor family [4, 26]. This design point favors a wider, shorter pipeline that, in practice, is clocked at a lower rate than competing designs from Intel. Since the register renaming scheme does not affect our experiments, all of the processor configurations use implicit renaming via the reservation stations (i.e., without an explicit register map table). Table 2-4 shows the details of all three configurations,

Table 2-4. Parameters of Target Systems for Online Diagnosis Evaluation

Feature	Narrow	Deep-Narrow	Short-Wide
pipeline stages	20	31	12
width: fetch/issue/commit/check	3/6/3/3	3/6/3/3	9/9/9/9
branch predictor	2-level GShare, 4K entries	2-level GShare, 4K entries	2-level GShare, 4K entries
instruction fetch queue	64 entries	64 entries	72 entries
reservation stations	32	32	54
reorder buffer	128 entries	128 entries	216 entries
load/store queue	48 entries	48 entries	44 entries
integer ALUs	3 units, 1-cycle	3 units, 1-cycle	6 units, 1-cycle
integer multiply/divide	1 unit, 14-cycle multiply, 60-cycle divide	1 unit, 14-cycle multiply, 60-cycle divide	1 unit, 8-cycle multiply, 74-cycle divide
floating point ALUs	2 units, 1-cycle	2 units, 1-cycle	3 units, 5-cycle
floating point multiply/divide/square root	1 unit, 1-cycle multiply, 16-cycle divide/square root	1 unit, 1-cycle multiply, 16-cycle divide/square root	1 unit, 24-cycle multiply, 26-cycle divide, 35-cycle square root
L1 I-Cache	16KB, 8-way, 64-byte blocks, 2-cycles	16KB, 8-way, 64-byte blocks, 2-cycles	64KB, 2-way, 64-byte blocks, 3-cycles
L1 D-Cache	16KB, 8-way, 64-byte blocks, 2-cycles	16KB, 8-way, 64-byte blocks, 2-cycles	64KB, 2-way, 64-byte blocks, 3-cycles
L2 cache (unified)	1MB, 8-way, 128-byte blocks, 7-cycles	1MB, 8-way, 128-byte blocks, 7-cycles	1MB, 16-way, 128-byte blocks, 20-cycles
Diagnosis: error counters	1249 bits	1249 bits	1219 bits
Diagnosis: FDU tracking	19 lines	19 lines	22 lines

Shaded entries for Deep-Narrow are identical to those of Narrow

including the overheads for our diagnosis scheme. We utilize the DIVA-style checker capability provided by sim-mase and additionally modified SimpleScalar to allow for hard fault injection.

We simulate the SPEC2000 CPU benchmarks, and we use the SimPoint toolset [63] to choose statistically representative samples of these long benchmarks for detailed simulation. We inject varying numbers of both stuck-at errors and coupling errors into the protected structures. Due to fault masking, injected hard faults do not always lead to errors when the faulty structures are

accessed. For example, a stuck-at-one fault does not effect a bit that is dynamically set to one during execution.

2.4.2 SRAS-CR and SRAS-EDC

In this section, we evaluate the benefits and costs of adding self-repair to microprocessor arrays. Our goal is to determine whether self-repair is viable, primarily in terms of performance, as performance is of critical importance in the commodity processor design space. We will compare both SRAS-CR and SRAS-EDC to systems protected with DIVA as well as to each other. Comparing SRAS to DIVA is somewhat unfair, since DIVA was not designed to handle hard faults, but it is the best alternative currently available. An important question we seek to answer is whether SRAS-EDC can achieve comparable performance to SRAS-CR despite not requiring DIVA support or the other drawbacks of SRAS-CR. While we compare performance quantitatively, the implementation costs and power consumption comparisons are qualitative.

First, we present the results of our evaluation of both SRAS-CR and SRAS-EDC. Our focus, detailed in the results and discussion that follows, is on comparing performance of fault-free microprocessors with performance of both fault-free and faulted microprocessors with SRAS in place. Another important factor to consider is the area of the processor core that we can effectively protect with SRAS. The percentage of the microprocessor core that SRAS protects depends on implementation. Specific details are proprietary, but estimation can be done with annotated die photos of a representative chip. Such analysis of the Alpha 21264 [64] shows that these array structures comprise roughly 33% of the non-cache microprocessor core die area.

2.4.2.1 Fault-Free Performance

Our first experiment explores the performance impact of SRAS for a system with no faults injected. The goal of this experiment is to determine the fault-free performance overhead of our

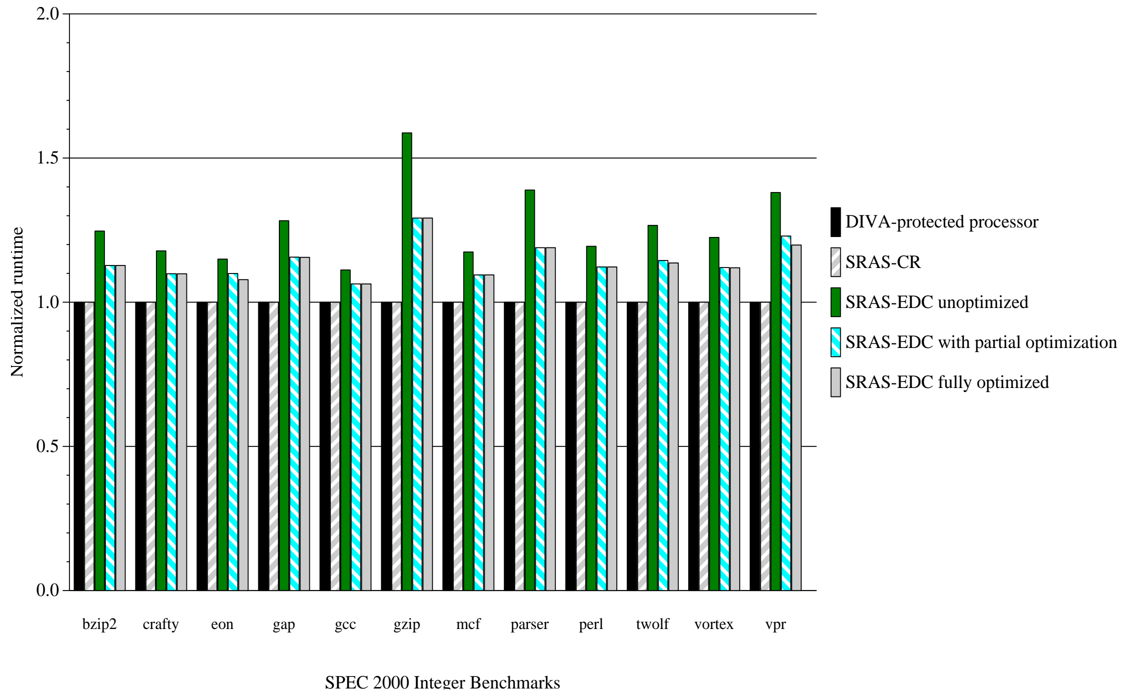
schemes relative to a system with DIVA. In Figure 2-7, we plot the fault-free runtimes (taller bars correspond to worse performance) of several systems, normalized to the baseline case of a system with DIVA (or an unprotected system), for all of the SPEC integer and floating point benchmarks. For each benchmark, we plot: (a) the baseline, (b) SRAS-CR (protecting just the ROB[2]), (c) unoptimized SRAS-EDC, (c) SRAS-EDC with a partial optimization in which we compute functional unit EDC in parallel for addition and subtraction, since these are the most common and the easiest to perform in parallel (i.e., they require the least extra hardware for parallel EDC computation), and (d) SRAS-EDC with the full optimization described in Section 2.2.6 to compute all functional unit EDCs in parallel (or not compute them at all, for self-checking circuits). The results show that SRAS-CR has the same performance as the baseline and that SRAS-EDC unsurprisingly incurs some penalty with respect to DIVA, due to adding some EDC logic on the critical path. The full optimization for SRAS-EDC helps quite a bit on most benchmarks, and the partial optimization does almost as well. One trend is that SRAS-EDC tends to suffer worse degradation in performance on the integer benchmarks, explained by the fact that the extra pipeline stages in SRAS-EDC exacerbate the branch misprediction penalty which is incurred more frequently by the integer benchmarks.

2.4.2.2 Performance in Presence of Faults

In this experiment, we study the performance benefit of self-repair for a system in which hard faults have been injected. Our goal is to determine whether self-repair provides enough benefit in the presence of hard faults to be worth its costs (in terms of implementation and fault-free performance). In Figure 2-8, we plot the runtime of SRAS-EDC versus that of a system protected by

---

2. Results discussed later will show that protecting the BHT is not worthwhile, and thus we do not wish to incur its fault-free performance penalty in this experiment.



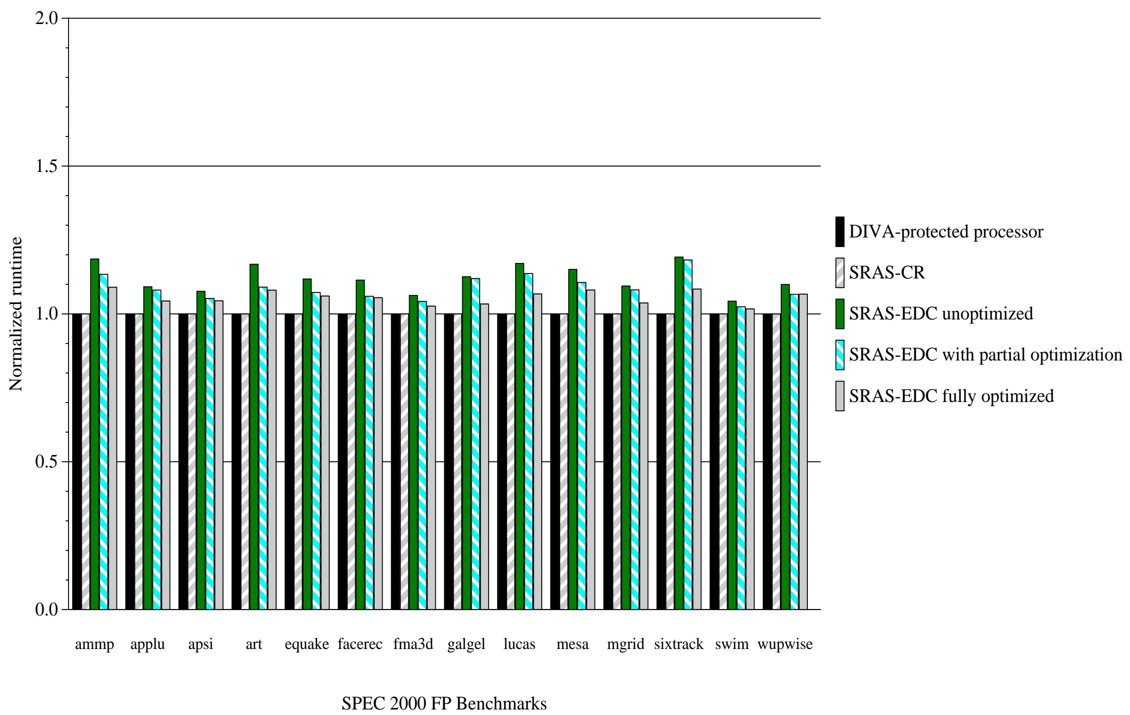
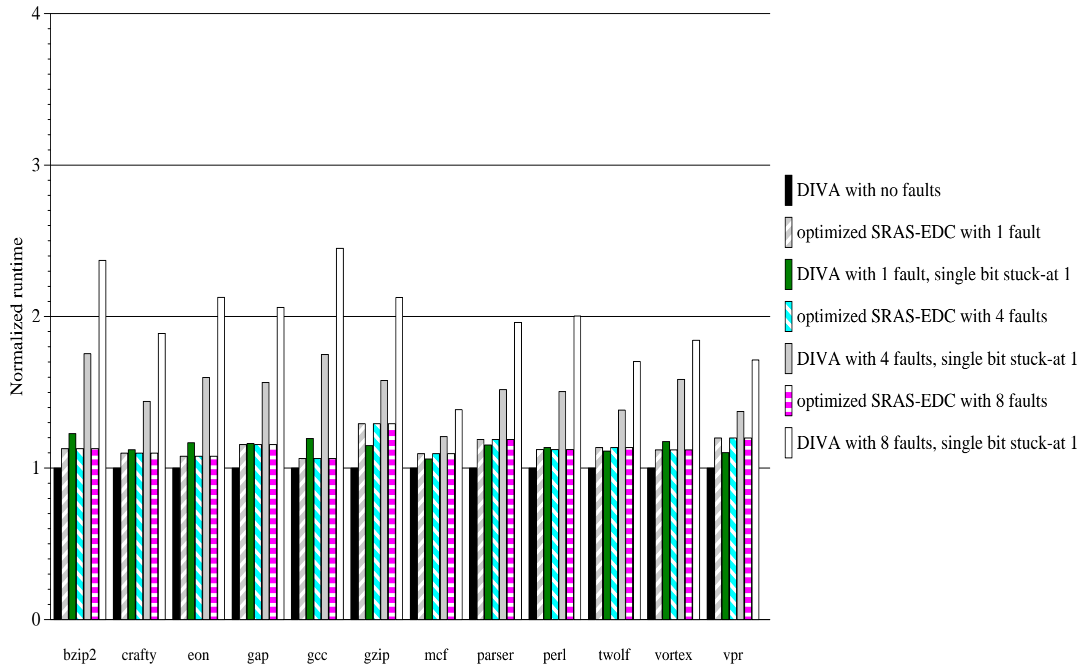


Figure 2-7. Fault-Free Runtime



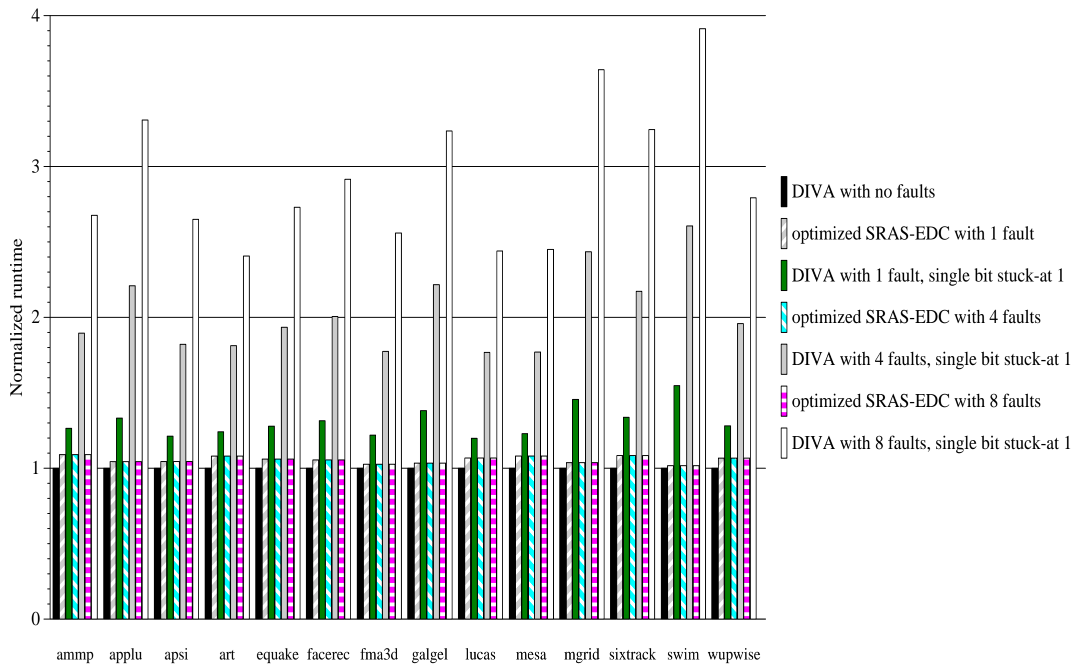
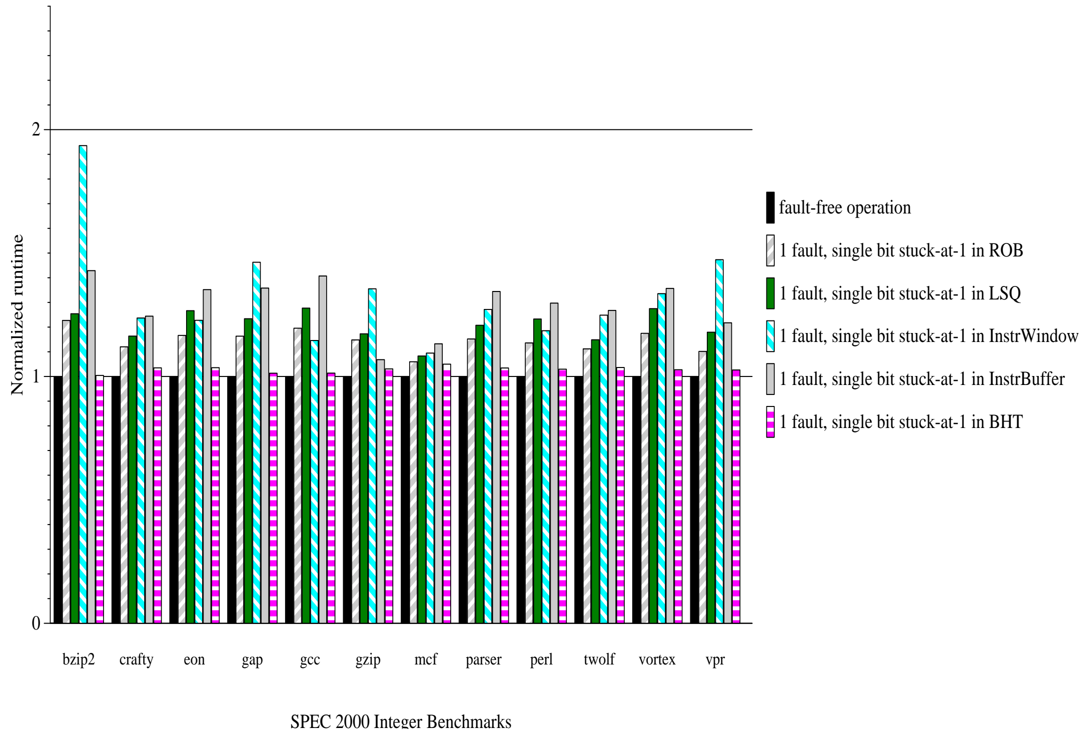


Figure 2-8. Runtime with Hard Faults Injected into the Reorder Buffer

DIVA, in the presence of hard faults injected into the reorder buffer. We do this to show that the SRAS-EDC fault-free performance costs are well-justified if hard faults are likely to be present over the lifetime of the part. Since SRAS-CR has no fault-free performance penalty, we do not include it in this plot (the bars would all show SRAS-CR equal to the bar labeled "DIVA with no faults"). We inject 1, 4, and 8 stuck-at-1 hard faults in order to evaluate the relative impact of varying numbers of hard faults. We normalize the results to the case of DIVA with no faults injected. Here we see that, in general, the presence of hard faults leads to SRAS-EDC outperforming DIVA. For the few integer benchmarks for which SRAS-EDC incurs the greatest fault-free performance degradation, however, DIVA may still have a slight advantage in the case of only one hard fault, but SRAS-EDC always outperforms DIVA for 4 and 8 faults. Considering that defect and fault rates are increasing, and we cannot eliminate all of them with burn-in testing [8, 56], these results demonstrate that SRAS is worthwhile. We observe that the floating point benchmarks derive relatively more benefit from self-repair. This effect is due to these benchmarks tending to better utilize the pipeline and thus incur more of a loss when an error causes DIVA to have to flush the pipeline.

2.4.2.3 Relative Performance Impact of Protecting Different Arrays

In this experiment, we explore the impact of hard faults on the other array structures that we are protecting with self-repair. Having shown in the previous experiment that ROB self-repair is beneficial in the presence of hard faults, we now compare the relative benefits of self-repair for other arrays. For each of the five structures we are protecting with self-repair—ROB, load-store queue, instruction window (scheduling window), instruction buffer (fetch buffer), and branch history table (BHT)—we injected a single stuck-at fault in that structure (i.e., we created five systems, each with a single fault in a different array). We then simulated each system's performance on a system with DIVA (i.e., without self-repair), to gauge the performance degradation that DIVA



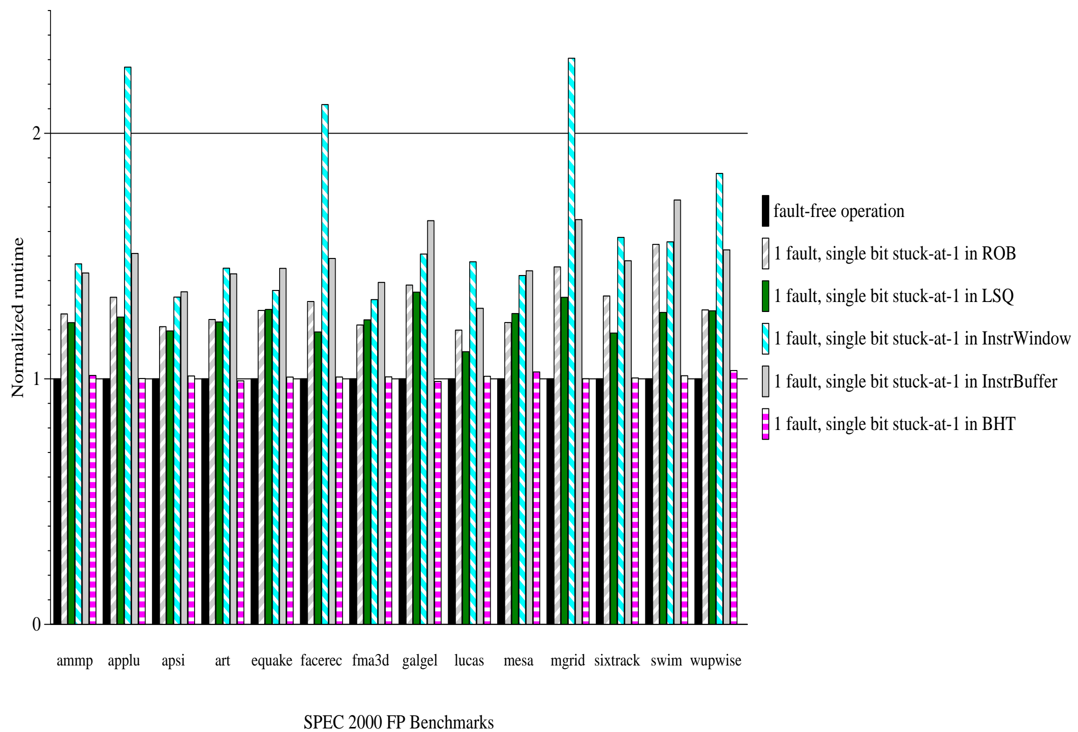


Figure 2-9. Impact on Runtime of Hard Faults on Other Array Structures

would incur (and that a system with self-repair would not incur). In Figure 2-9, we plot the runtimes for these five systems, normalized to a fault-free system. Thus a taller bar in the graph indicates that self-repair is more important for this structure. We observe that there is no one particular structure that is always the most important to protect with self-repair, although there are some patterns. For example, the instruction window benefits more from self-repair than the ROB, as does the instruction buffer for most benchmarks. A significant result is that faults in the BHT have virtually no impact on performance. This is because the BHT is a large structure that is accessed sparsely, and faults in the BHT are likely to be masked. Moreover, faults in the BHT can only lead to incorrect branch predictions, not incorrect execution, so the corresponding pipeline squashes can be initiated earlier (after the execution stage, instead of at the commit stage) and thus incur less performance penalty.

2.4.2.4 Implementation Costs

It would be unfair to favorably compare the implementation costs of SRAS against DIVA, since DIVA is mainly targeting a different problem (i.e., transient faults) and it can also tolerate hard faults beyond just the array structures (albeit with performance and energy penalties). A comparison of SRAS-CR and SRAS-EDC is reasonable, though. SRAS-CR requires DIVA (or some similar dynamic verification scheme) as an error correction backstop before a fault is determined to be hard, which is a significant cost for systems that would not have otherwise chosen to use DIVA. Besides needing DIVA, SRAS-CR adds dedicated check rows to each array for performing error detection/diagnosis. Moreover, SRAS-CR adds extra ports to the arrays in order to perform error detection/diagnosis (by writing and reading the check row and operational row to compare them). SRAS-EDC, unlike SRAS-CR, adds EDC bits to array entries. SRAS-EDC also adds EDC computation logic and EDC check logic at certain points in the pipeline. Both SRAS techniques provide

what we believe to be a low-cost alternative to traditional large-scale replication with better performance than a low-cost BER technique in the presence of hard faults in frequently accessed structures. The question of which SRAS technique is better has no definitive answer. For designs requiring the highest fault-free performance, SRAS-CR is better. SRAS-EDC, however does provide an implementation that does not require an additional fault detection mechanism to operate, which gives it the advantage of extra flexibility in its application.

2.4.3 Online Diagnosis

Our evaluation of our fine-grained online diagnosis mechanism consists of experiments to explore the effectiveness of our diagnosis scheme in a representative sample of processor designs. Our evaluation has the following goals:

- First, we want to show that commodity design points using our reliable architectural extensions can quickly and correctly detect and diagnose hard faults, even in the presence of transient faults.
- Second, we want to demonstrate that, after our scheme deconfigures a permanently faulty FDU, the microprocessor's performance is still good enough to be useful.
- Third, we want to compare our scheme against a microprocessor that simply relies on DIVA checkers to tolerate hard faults; while DIVA was designed primarily for soft faults, it can also tolerate hard faults, and we want to determine if our scheme outperforms this simpler solution.
- Fourth, we want to perform a sensitivity analysis for singleton complex, combinational logic units such as the integer and floating point multipliers in order to determine if protection of these units warrants further investigation.
- Finally, taking all three of our chosen design points together, we show the general applicability of the technique to a broad set of designs from the commodity microprocessor design space.

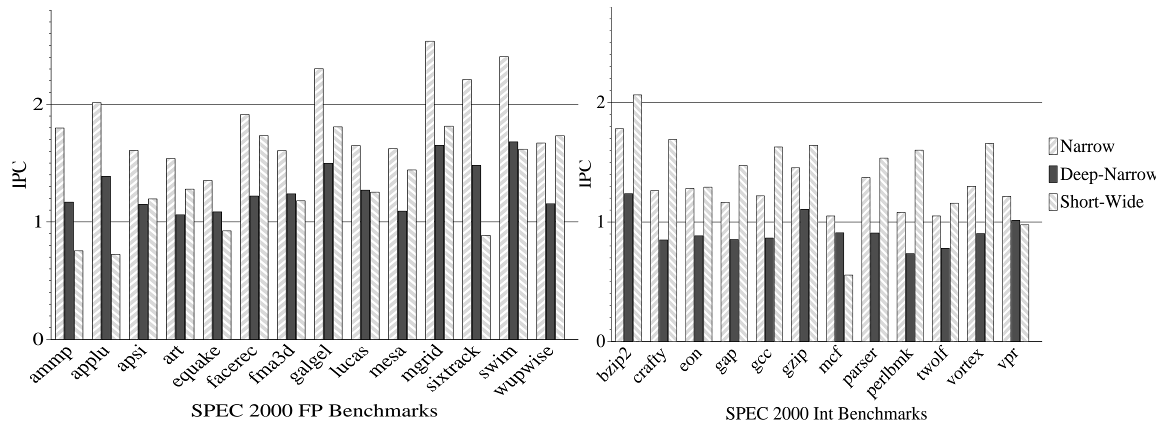


Figure 2-10. Error-Free Performance (SPECfp and SPECint) for Each of the Three Evaluated Processor Configurations

Since we present results in the rest of this section in terms of normalized performances, we provide baseline error-free IPC results for each of the three processor design points in Figure 2-10.

The goal of all of our evaluation is to show how the processor behaves in the presence of a hard fault. The likelihood of a hard fault affecting processor operation is highly dependent upon the process used to manufacture the part, the complexity of the design, and the operating environment that the part is deployed in. The discussion of these issues is an active body of research and is beyond the scope of this evaluation.

2.4.3.1 Detection and Diagnosis of Hard Faults

Our first set of experiments explores how accurately and quickly our scheme detects and diagnoses hard faults. In each experiment, we injected one hard fault in a single structure. All injected hard faults manifest as a single bit stuck-at-1. To accurately account for masking effects, we inject the hard fault at a specific site in the FDU, with the exception of complex FDUs for which we lack a detailed implementation. Our hard fault selection attempts to provide greater masking of fault effects, which leads to a smaller performance penalty and longer diagnosis latency due to fewer

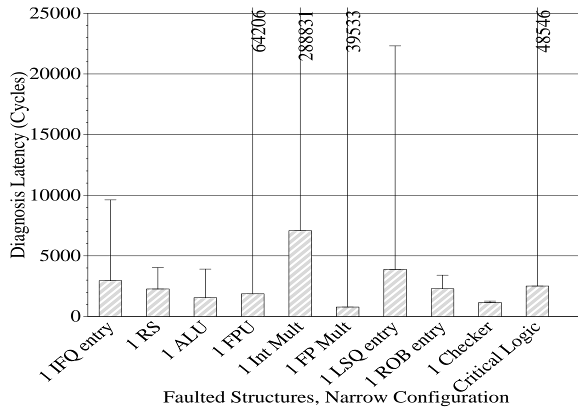


Figure 2-11. Hard Fault Diagnosis Latency, Averaged Over All Benchmarks, for Narrow Configuration

Error bars show one standard deviation for diagnosis latency above the average.

error detections and corrections. We do this because it is a pessimistic case for the operation of our mechanism.

Because transient faults are relatively rare for the intervals we are simulating, we expect no more than one transient to occur during a diagnosis interval. To model the effects of this scenario, we ran each simulation with the effect of a single, observed transient added at the beginning of the simulation (that is, one random set[3] of FDUs' counters started with an error count of one, rather than zero at the beginning of diagnosis). We observed no difference in the behavior of the diagnosis algorithm for these experiments, leading us to believe that the mechanism is robust in the presence of typical transient faults.

In order to accurately account for masking effects in our simulation environment, we extended SimpleScalar to include detailed simulation of the fault sites we inject errors at. To avoid excessive

3. A set is defined as one of each required FDU type for a particular instruction's processing. Recall that DIVA cannot determine whether an error came from a transient or hard fault and also cannot diagnose a fault's source, requiring the diagnosis mechanism to treat all errors detected by DIVA in the same fashion, with the counters for all FDUs involved in the calculation of the erroneous result getting incremented upon DIVA correction. For example, for an integer add instruction, a set would include critical logic, one integer ALU, one reservation station, one ROB entry, one IFQ entry, and one checker.

simulation times, we extended SimpleScalar only in the areas required to sufficiently evaluate the effects of masking for the injected fault. Fault sites were chosen for each of the FDUs in the system with the goal of providing a representative fault for the given structure, with nominal or slightly pessimistic behavior sought to ensure that our study would apply for the broader set of possible faults that could occur in the system.

For storage structures, we selected a representative bit to corrupt for a faulted unit. For the ROB, we inject the fault into the least-significant bit (LSB) of the data result. This causes the common value of 1 to provide data masking for the injected fault. For the RS and IFQ, we corrupt the LSB of the register identifier for the second argument of the instruction. This causes single-argument instructions to functionally mask this error and gives an even probability that two-argument instructions will experience data-masking for the injected fault. For the LSQ, we inject the fault in bit 16 of the address. This prevents data mis-alignment exceptions and provides an average-case data masking scenario.

For combinational logic units, such as the ALUs, corrupting a single bit of output is not an accurate fault model. This is due to the fact that combinational logic differs from storage in that faults may propagate to different outputs or may be functionally masked for different inputs and operations. This requires us to either simulate a gate-level design of the faulted unit or to utilize a statistical fault model.

For the integer ALUs, we model faults as manifesting in the adder. We used a gate-level design for a 32-bit adder and selected a representative gate whose output is stuck-at-1 when the fault is injected. We performed a thorough gate-level fault simulation of the adder. We then simulated all possible inputs and all possible fault locations for the adder to gain intuition on how masking affects observation of fault effects. The gate we selected for fault injection in our simulations rep-

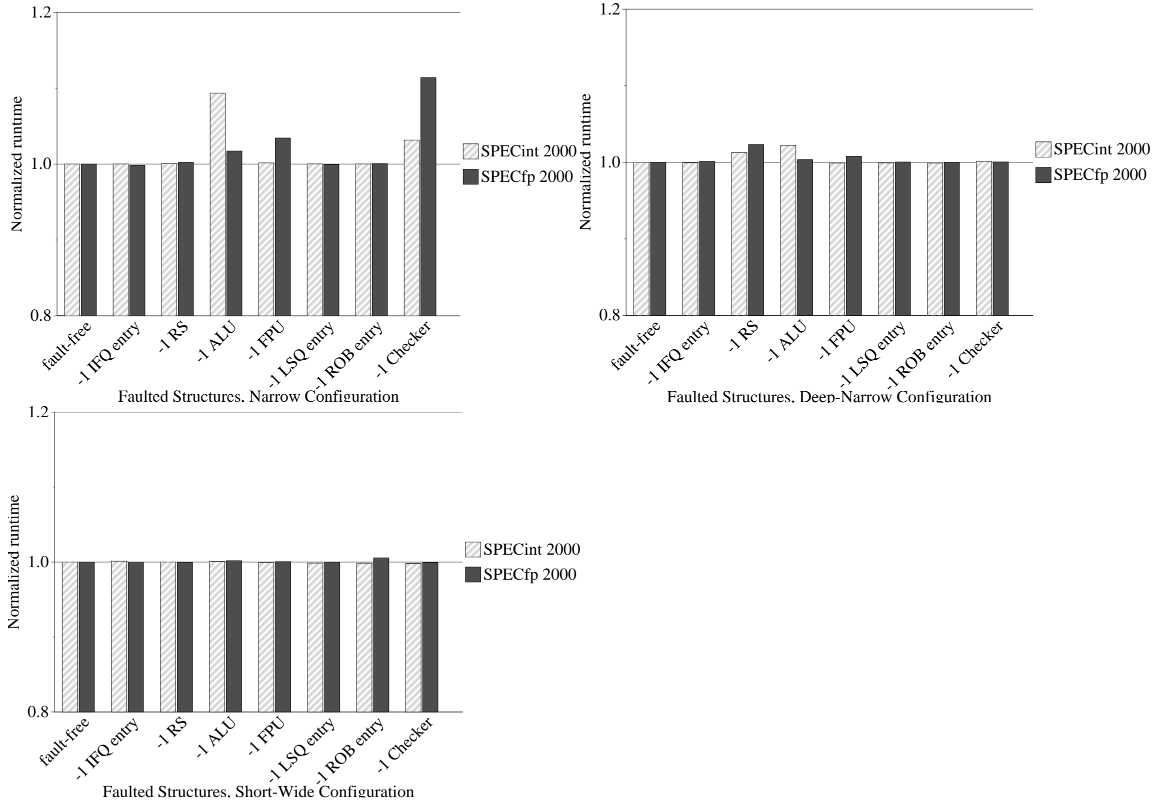


Figure 2-12. Performance Impact of Losing One Component to a Hard Fault for Each of the Three Evaluated Processor Configurations

resents the nominal masking case with a shading toward more masking, as this is a pessimistic assumption in our experiments. Masking was then evaluated for every instruction that accessed the ALU with the faulty adder.

For the integer multiplier, floating point multiplier, and floating point ALUs, we used a statistical model for fault injection. In this model, we assume that there is a 50% chance that data masking will mask the injected fault. We use a random number generator to select which instructions observe this data masking effect.

In all of our experiments, the microprocessor detected and diagnosed the injected hard fault and did not mis-diagnose a soft fault as being hard. We measured how many cycles elapsed before an injected hard fault was correctly diagnosed, and we plot the results of this experiment for the worst

Table 2-5. Number of Diagnoses Needed to Identify Correct Failing Unit

Faulted Unit	1 Diagnosis	2 Diagnoses	3 Diagnoses	4 Diagnoses	5 Diagnoses	6+ Diagnoses
instruction fetch queue entry	>99.99%	<0.01%	0%[a]	0%[a]	0%[a]	<0.01%
reservation station	>99%	<2%	<0.01%	<0.01%	<0.01%	<0.1%
integer ALU	>99%	<0.1%	<0.1%	0%[a]	0%[a]	<0.1%
floating point ALU	>99%	<0.1%	<0.1%	<0.01%	<0.1%	<0.1%
integer multi-plier	>99.999%	0%	0%[a]	0%[a]	0%	0%[a]
floating point multiplier	>99.99999%	0%	0%	0%	0%[a]	0%
load/store queue entry	100%	0%	0%	0%	0%	0%
rob entry	>99.99%	0%[a]	0%[a]	0%[a]	0%[a]	0%[a]
DIVA checker	>99%	<1%	<0.01%	0%[a]	0%	0%[a]
critical logic	>94%	<4%	<1%	<1%	<1%	<1%

a. Value less than 0.001%, but non-zero value.

of the three configurations (Narrow) in Figure 2-11. The other two configurations exhibited qualitatively similar performance, so are not shown here. Since the results were relatively insensitive to the benchmarks, we present the mean results for the entire SPEC2000 benchmark suite; the error bars in the figure represent one standard deviation above the mean. The results show that most hard faults are diagnosed within fewer than 15,000 cycles, but that there are irregular diagnoses that take significantly more time, leading to a high variance in the data. These irregular diagnoses come from two sources.

The first source is initial mis-diagnosis of non-faulty hardware. To gain intuition on how often this will be a factor in diagnosis latency, we gathered statistics on how many diagnoses are required before converging on the correct diagnosis. In these simulations, the fault was always left active, allowing for continual diagnosis of the same faulty unit. Table 2-5 shows the results of

these experiments. Because the results for all processor configurations are similar, we combine them in the data presented. While only the load/store queue entry has perfect diagnosis across all configurations, all units except critical logic are diagnosed initially with at least 99% accuracy. With critical logic, the fact that multiple units get deconfigured before the correct problem is identified is unimportant because a fault in the critical logic will require that the processor be shutdown. As the latency data in Figure 2-11 shows, this still happens in a very short period of time. In effect, the counter threshold selection for critical logic allows the greatest opportunity for correct diagnosis of an FDU prior to drawing a conclusion that critical logic has been affected by a hard fault. Since the reaction to such a hard fault is more drastic than deconfiguring a single FDU, we feel that this is a wise design decision.

The second source of variance in diagnosis latency is programmatic phase behavior. The mix of instructions varies throughout the various phases of program operation. During certain phases, FDU utilization patterns will shift, causing diagnosis behavior to vary. In rare circumstances, a string of instructions that causes the wrong error counter to saturate first will occur (for example, a loop that repeats many times). This can lead to a large number of mis-diagnoses before the faulted unit gets properly deconfigured. As mentioned previously, the diagnosis mechanism tolerates these mis-diagnoses without significant impact to the performance of the processor. The largest observed latencies were on the order of millions of cycles, which is a small amount of time for a modern microprocessor running at multiple-gigahertz clock frequencies.

Our diagnosis latency study shows that the window of vulnerability for a faulty DIVA checker is, on average, around 2,000 instructions, which is easily within the recovery capabilities of typical hardware and software backward error recovery (BER) mechanisms. The different diagnosis latencies for different FDUs are a function of the relative usages of these structures as well as their error

counter thresholds. Nevertheless, for all structures other than the DIVA checkers, the diagnosis latency is relatively unimportant, since between when the fault occurs and when it is diagnosed and the FDU deconfigured, the checkers mask its effect with only a performance penalty caused by the number of pipeline flushes equal to the error counter threshold for the faulty FDU. Over the course of even thousands of cycles, this performance penalty is still negligible. The key is not incurring that performance penalty over the entire lifetime of the processor, as results in Section 2.4.3.3 show.

For the microarchitectures in our experiments, there are no spare units for the integer multiplier or floating point multiplier. Thus, we are unable to evaluate the effects of deconfiguring these units in Section 2.4.3.2, because they are essential to correct operation of the processor. The latency and accuracy data do suggest that considering these units as FDUs is possible. In Section 2.4.3.3, we show that protecting these units from hard faults with a diagnosis and deconfiguration strategy is worth considering in future designs.

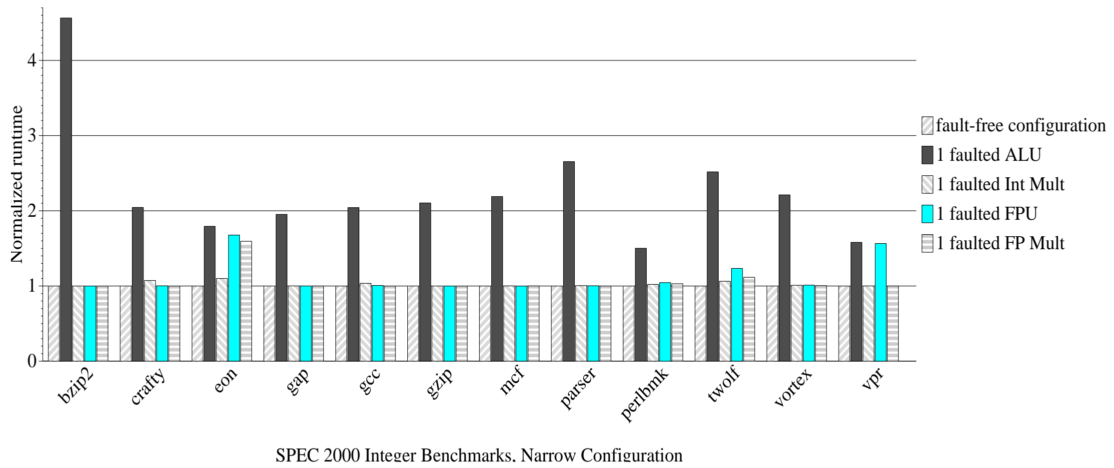
2.4.3.2 Performance After Deconfiguring FDU

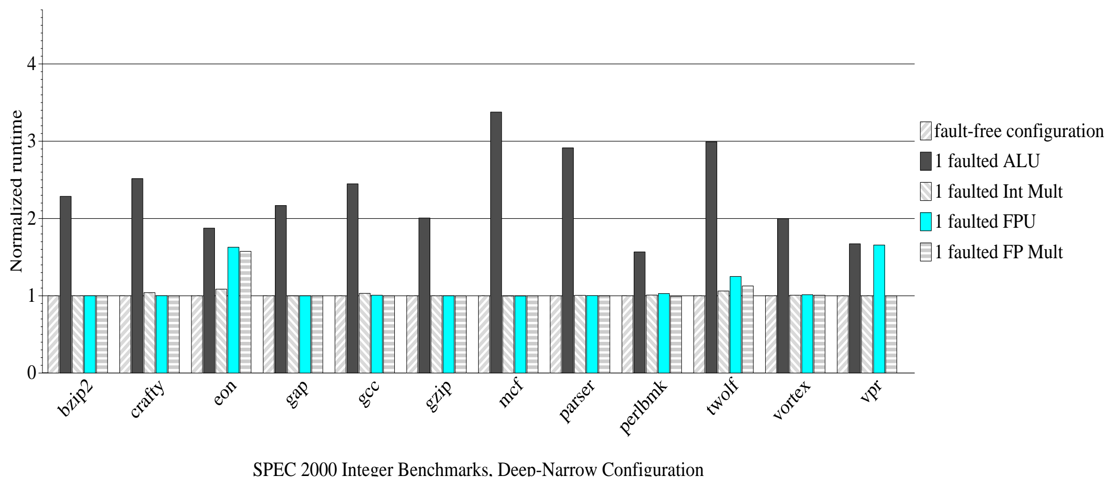
The second set of experiments evaluates the performance impact of de-configuring an FDU after having diagnosed it as being permanently faulty. In each of these experiments, we remove one of each type of FDU that we study. Figure 2-12 plots the runtime for each of these experiments, normalized to the error-free (fully-configured) case. Since there is little variation in the results across benchmarks, we plot the average results (geometric means of normalized runtimes) across the SPECint and SPECfp benchmarks for each processor configuration. The data show that the performance impact of deconfiguring an FDU is often small. This result, which corroborates prior work [64, 69], is in part due to the fact that the processor configurations we are modeling are over-provisioned for single SPEC benchmarks; both of the Pentium 4-styled configurations (Nar-

row and Deep-Narrow) are designed to run multiple threads simultaneously, and the extreme width of the Athlon-styled configuration (Short-Wide) has it provisioned with multiple units. Thus, resources are often idle in a typical single-threaded workload. There is a non-negligible performance degradation due to deconfiguring an ALU or DIVA checker in the Narrow configuration. This penalty all but disappears in the other two configurations. In Deep-Narrow, the longer pipeline suffers more from pipeline flushes, which degrade performance to a point where the performance loss of the execute and commit bandwidth is effectively masked. In Short-Wide, the extra units provisioned to support the width of the processor effectively mask the penalty for removing a single unit. Stated another way, removing a single unit in Short-Wide is removing a smaller percentage of available computing bandwidth than in the Narrow configurations. All of these faulty systems continue to function correctly and with reasonable performance.

2.4.3.3 Performance with Just DIVA Recovery (But No Diagnosis)

In this last set of experiments, we evaluate the performance of a microprocessor that relies strictly on the DIVA checkers to tolerate hard faults. While DIVA was designed primarily for soft faults and thus this is not a basis for a perfectly fair comparison, DIVA can tolerate hard faults and it is instructive to compare against this option. A DIVA-only system is also similar to a system that uses redundant threads for error detection and flushes the pipeline to recover from errors (assuming forward progress can be ensured). Figure 2-13 and Figure 2-14 show the effects of allowing complex, combinational logic sub-structures with hard faults to remain in use with the DIVA checkers correcting the errors that they activate for the SPECint and SPECfp benchmarks, respectively. Figure 2-15, for SPECint, and Figure 2-16, for SPECfp, show the effects of allowing regular array structures with hard faults to remain in use with only DIVA correction. In all four figures, we plot runtimes that are normalized to the error-free case for each configuration, but we do not





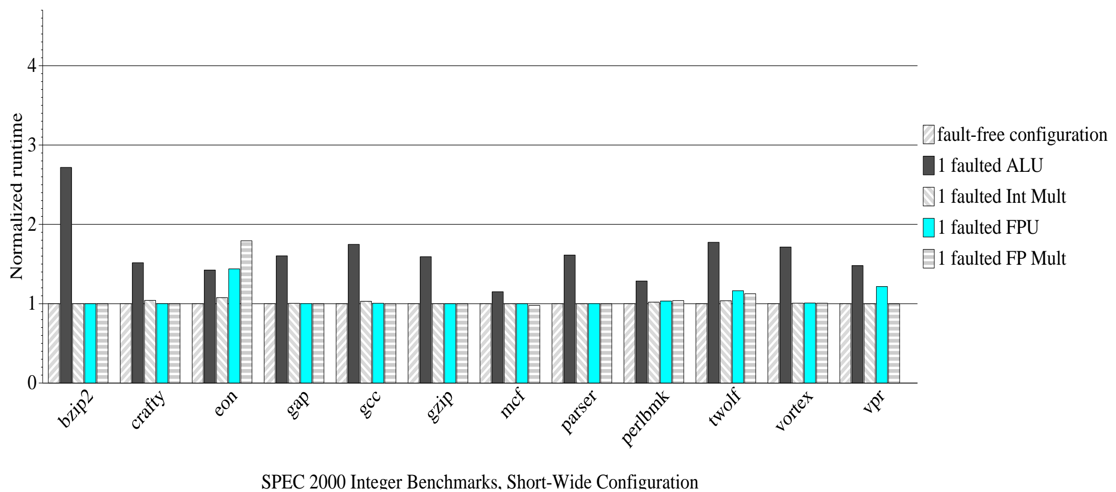
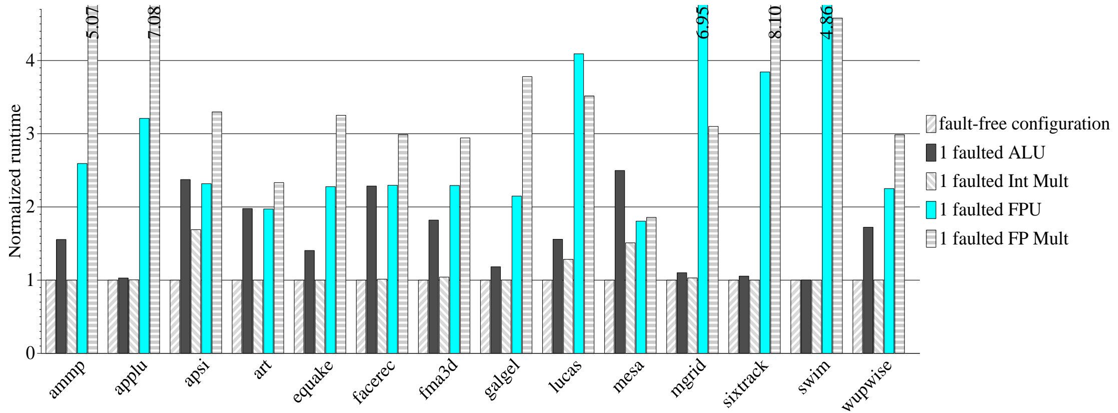
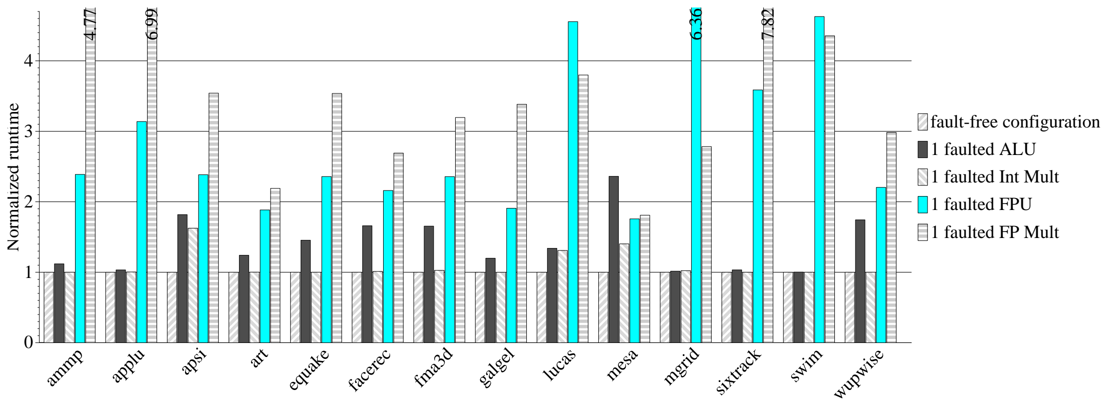


Figure 2-13. Performance of DIVA-Only Correction for Combinational Logic Units (SPECint)





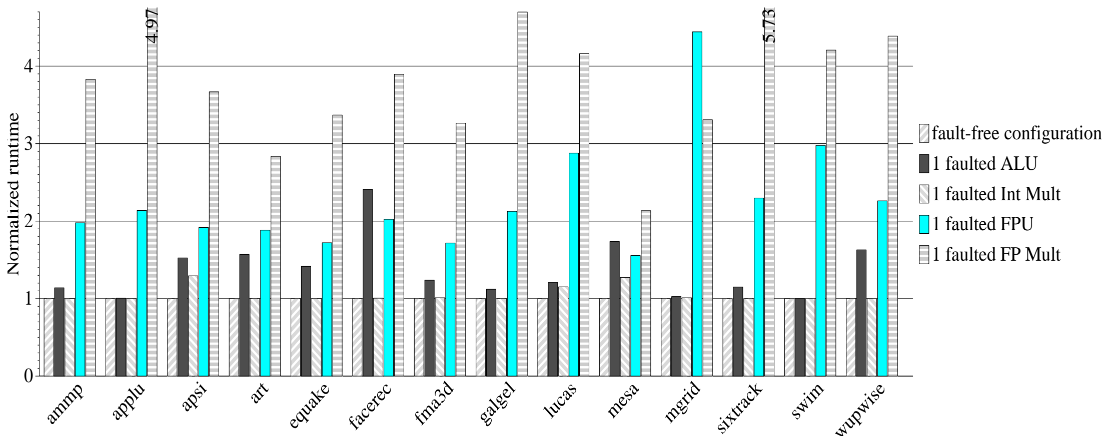
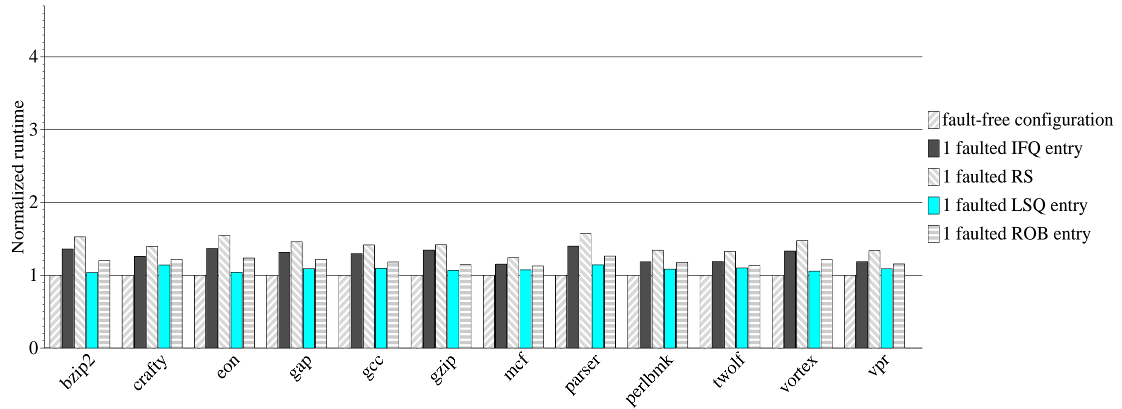
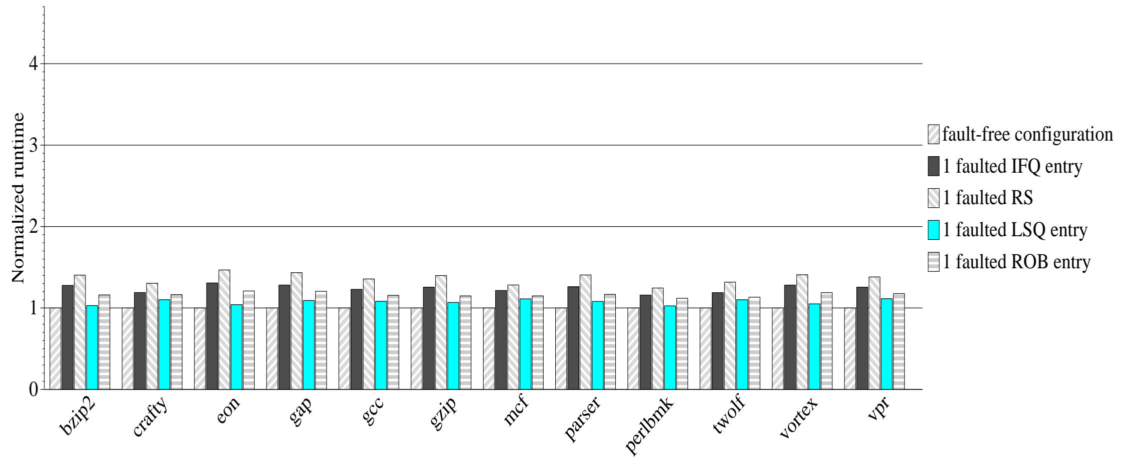


Figure 2-14. Performance of DIVA-Only Correction for Combinational Logic (SPECfp)





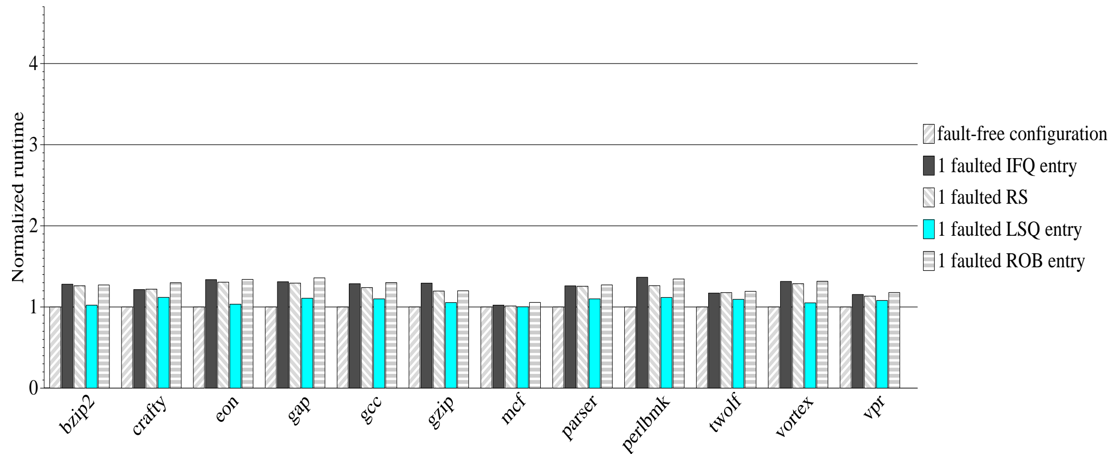
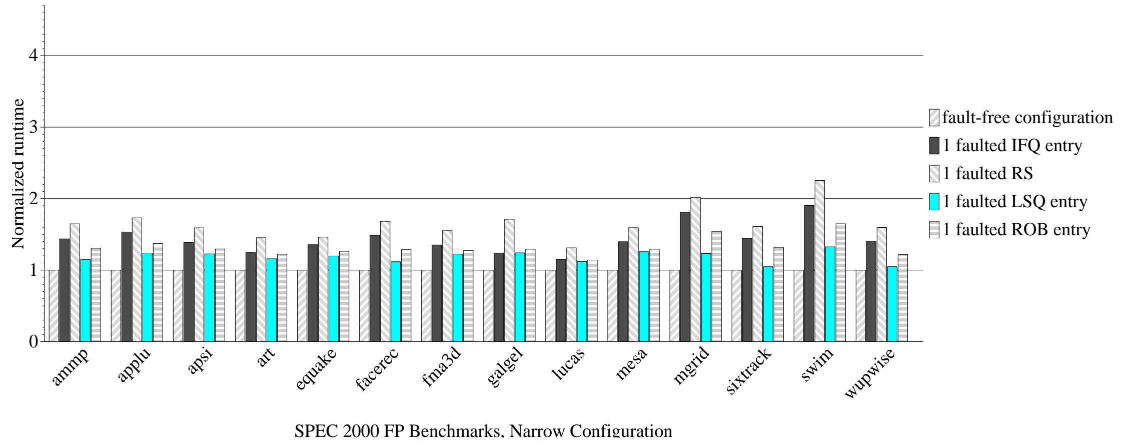
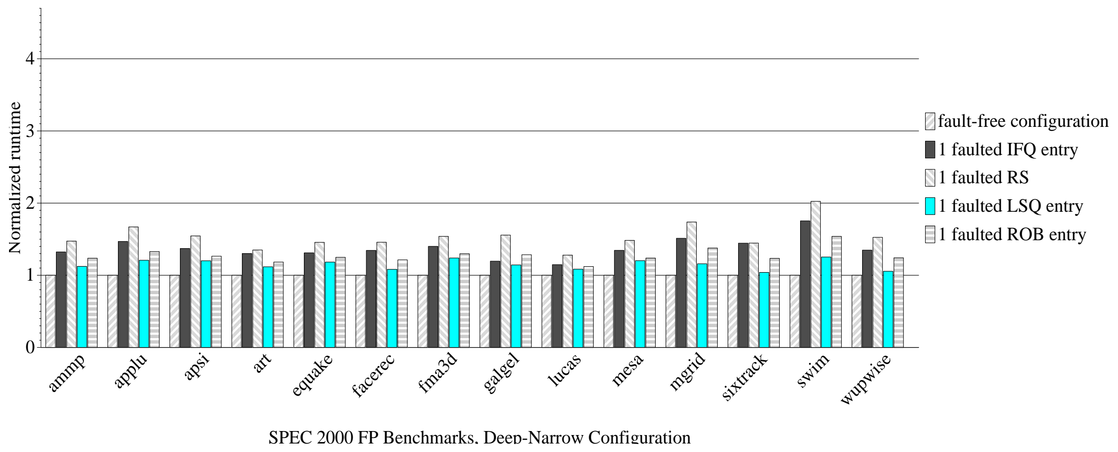


Figure 2-15. Performance of DIVA-Only Correction for Array Logic Units (SPECint)





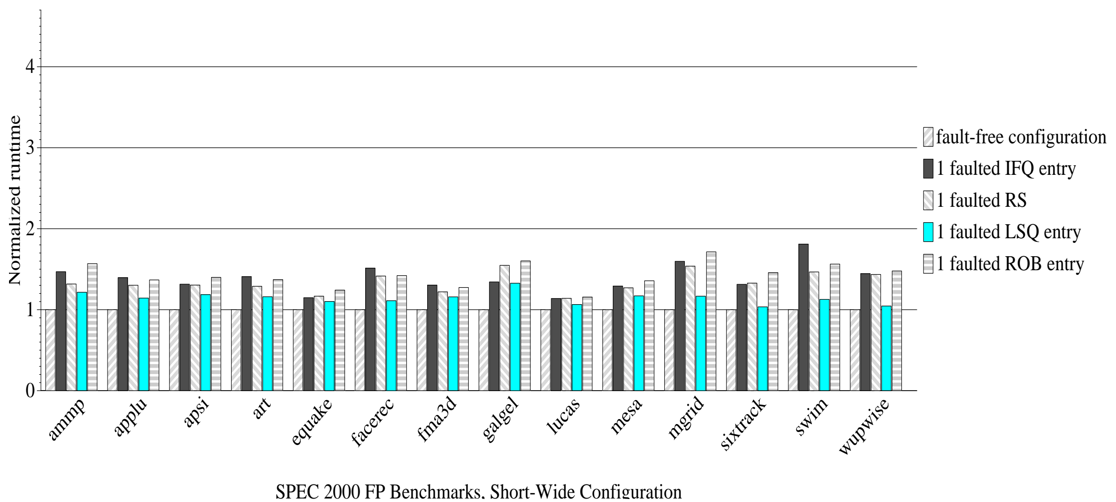


Figure 2-16. Performance of DIVA-Only Correction for Array Logic Units (SPECfp)

aggregate results across benchmarks because there is significant variability across benchmarks. In these figures, the bar order, from left to right, matches the order of items in the legend, from top to bottom, with a full set of bars provided for each of the SPEC2000 benchmarks. We do not inject hard faults into the DIVA checkers because they cannot tolerate them without our diagnosis/reconfiguration.

In the case of the complex combinational logic units, the structures into which we are injecting faults are used frequently and are critical to the correctness of the processor. The results show that hard faults have a drastic impact on system performance when DIVA is forced to correct the errors they create. The performance of the DIVA-only system is far worse than the performance we demonstrated for our system in Section 2.4.3.2. Technology trends toward deeper pipeline implementations will only serve to make the performance penalty for each error's recovery (i.e., pipeline flush) more severe. The data for the singleton units in our study (the integer and floating point multipliers) shows that, for certain workloads, there is motivation to provide a less-costly alternative to pipeline-flushing error correction mechanisms.

For the array structures, there are many more units present in typical architectures than there are combinational logic units. Because of their greater population in modern designs, these units are naturally used less often than the combinational logic units. This functional masking effect results in the lessened effects we observe. These units are still used often enough to cause frequent pipeline flushes from DIVA corrections to noticeably, negatively impact performance.

The relative difference in magnitude of the structure-to-structure penalty is directly related to how frequently a given sub-structure is used by the workload. Benchmark-to-benchmark variation for a given type of FDU is a result of the distribution and frequency of pre-existing stall events in a given benchmark. The causes of these events, such as cache misses or branch mispredictions,

result in a percentage of corrected errors falling in the shadow of another pipeline-clearing event, thus diminishing the penalty associated with the error correction. For example, a benchmark with many branch mispredictions is less sensitive to pipeline flushes due to errors, if the errors tend to occur soon after branch mispredictions, since there is less state that gets flushed by the error.

2.4.4 Summary and Discussion of Results

In this work, we have presented a framework for designs for self-repair of microprocessor array structures, and we have developed two particular designs based on that framework: SRAS-CR and SRAS-EDC. These designs are motivated by the belief that per-part hard fault rates will increase as we scale CMOS to smaller and smaller device geometries and pack ever more devices into a single microprocessor. This motivation is grounded in cautionary statements from the ITRS [28] and detailed studies of lifetime reliability by Srinivasan et al. [68].

A survey of methods for achieving hard-fault tolerance in the microprocessor core shows that we have a gap in capability for protecting the non-cache area (the processor pipeline). This gap stems from the following two facts:

1. Traditional hard-fault tolerance design points could afford large-scale redundancy, for example replication of the entire core, so they employed techniques like TMR to achieve fault tolerance

2. Newly-developed low-cost fault-tolerance techniques are not designed to tolerate hard faults—even though they sometimes can, this tolerance comes at a high performance cost.

In the commodity microprocessor market, performance and cost are the key motivating constraints. As the aforementioned CMOS trends begin to impact this design space, we believe that fault tolerance will gain in importance. Low-cost methods to achieve hard fault tolerance will become necessary as a result. The two SRAS methods presented are two such designs.

In the case of fault-free execution, both SRAS methods may add some performance overhead compared to an unprotected system, due to the few instances in which self-repair logic is on the critical path. However, if hard faults exist in arrays, then SRAS outperforms the existing light-weight approaches for tolerating faults while avoiding large-scale replication of microprocessor cores. As hard fault rates continue to increase, we believe that SRAS will become an increasingly attractive design point.

To address the emerging problem of operational hard faults and fabrication defects in microprocessors, we have developed a microprocessor design that leverages the existing redundancy in current microprocessors. This redundancy, which exists to improve performance by exploiting ILP and thread level parallelism, can be used to mask hard faults. Our microprocessor design integrates DIVA-style error detection with a new mechanism for diagnosing hard faults. After diagnosis, it de-configures the faulty FDU and continues operation. Experimental results demonstrate that our scheme can accurately and quickly diagnose hard faults and reconfigure around faulty FDUs to provide a microprocessor that performs only somewhat worse than a fault-free system.

As technology trends continue to drive higher-complexity designs, implemented with smaller transistor geometries, we believe that the incidence of hard faults will increase, both from manufacturing defects and lifetime wearout effects. In response to this increase in hard faults, commodity microprocessor designs will require that hard fault tolerance be considered in their designs. Traditional approaches in the fault-tolerant computing space have not been limited by the same cost constraints as the commodity space, making direct application of existing techniques inappropriate. We believe that the commodity microprocessor design space will drive the following constraints into a fault-tolerant design:

- Low-cost implementation in terms of hardware and power consumption characteristics.

- Graceful degradation in performance in the presence of hard faults.
- Effective containment of lifetime-reliability induced defects.

To meet these constraints, fine-grained diagnosis schemes will be required, since coarse-grained solutions tend to incur too much performance penalty per fault tolerated. The present online techniques will have to be adapted to work in concert with existing features in the commodity design space, including low-cost error detection and correction mechanisms.

The experimental results in this section confirm that existing microprocessors have redundancy that can be exploited to tolerate hard faults. We have also shown that, for a variety of processor configurations, we can accurately and quickly diagnose hard faults and reconfigure around faulty FDUs to provide a microprocessor that performs only slightly worse than a fault-free microprocessor. Moreover, it vastly outperforms the alternative of just relying on DIVA.

Technological and architectural trends drive this work and encourage further work in this area. The incidences of hard faults and fabrication defects will continue to increase. This will lead to decreased yield, higher FIT rates, and lower MTTF for future generation parts. We have shown that use of a diagnosis and deconfiguration mechanism will allow for parts to operate in the presence of hard faults until they begin to experience larger numbers of hard faults near their end of life. This will lead to higher MTTF/lower FIT rates for parts that use this sort of scheme over their unprotected peers. Also, as microarchitects try to exploit ever more ILP and thread level parallelism, there will be even more redundancy that can be leveraged for improving reliability and yield. In particular, emerging SMT processors will have more redundant hardware and fewer singleton resources. Thus the advantages of our approach will increase due to these trends. The caveat is that, as workloads evolve to take advantage of this extra hardware, the performance impact of having to deconfigure an FDU will increase. If that is the case, cold sparing of performance-essential

FDUs may be employed to effectively increase the MTTF/decrease the FIT rate of a part employing our scheme. As mentioned previously, quantitative analysis of how much MTTF/FIT rate improvement will be gained is dependent upon fault rates, which are dependent upon process and design details that we do not consider in this work. Nevertheless, even without cold spares, a heavily loaded microprocessor will continue to function correctly and with better performance than just DIVA in the presence of operational hard faults and fabrication defects.

2.4.5 Related Work

In this section, we present prior research in tolerating hard faults and fabrication defects. A canonical design for tolerating hard faults is the IBM mainframe [66]. Mainframes not only have redundant processors, but they also incorporate redundancy within the processor in order to seamlessly tolerate hard faults. The IBM G5 microprocessor, for example, has redundant units for fetch/decode and for instruction execution. Some other traditional fault-tolerant computers, such as the Stratus [82] and the Tandem S2 [31], simply replicate entire processors. An even more extreme case of using redundancy to tolerate fabrication defects and, to a lesser extent, operational hard faults, is the Teramac [20]. The Teramac is designed to make use of components that are likely to be faulty, and it is motivated by expected defect rates in nanotechnology. While these systems all provide excellent resilience to hard faults, such heavyweight redundancy incurs significant costs in terms of hardware and power consumption.

DIVA [6] and redundant thread schemes provide low cost and low power alternatives to heavyweight redundancy. All of the redundant threading schemes (AR-SMT [58], Slipstream [71], SRT [47, 55], and SRTR [78]) provide error detection and either use pipeline squashing for error correction or could easily provide error correction via pipeline squashing. All of these schemes were designed for transient faults and thus share the same drawback as DIVA, with respect to hard faults, since they incur a pipeline squash (and its corresponding performance and energy penalty)

every time a fault manifests itself. For hard faults in frequently-used microprocessor structures, fault manifestation is too frequent and the performance of these schemes suffers.

There are lightweight approaches by Aggarwal et al. [2, 3], Shivakumar et al. [64] and Srinivasan et al. [69] that, similar to our work, leverage existing redundancy in microprocessors. Aggarwal et al.'s work differs in that it treats the core as a field-deconfigurable unit and explores opportunity to exploit on-chip redundancy for the core and other structures outside of the core, such as memory controllers. Shivakumar et al.'s work differs in that it is strictly for tolerating fabrication defects and does not extend to hard faults that occur during execution. They combine offline (pre-shipment) testing and diagnosis of microprocessors with deconfiguration capabilities to improve effective yield. Our approach combines deconfiguration with online error detection and fault diagnosis to improve both yield and reliability. Srinivasan et al.'s work does not address error detection or fault diagnosis.

An approach to improving microprocessor reliability in the presence of operational hard faults (but not fabrication defects) is to use dynamic reliability management [67]. In this approach, the processor dynamically adapts, based on a model of its estimated lifetime, in order to achieve a desired lifetime. In particular, if the processor is running too hot, due to a particular workload, it may use dynamic voltage scaling to cool down and improve its reliability. This approach is orthogonal and complementary to ours.

Another scheme for tolerating only fabrication defects, called Rescue [61], utilizes circuit transformations to improve testability and enable coarse-grain diagnosis of defective components (ways of a superscalar processor). The finer grain diagnosis in our research enables us to discard less fault-free hardware, and it may enable us to tolerate more hard faults before failure.

There are other non-comprehensive approaches to tolerating hard faults in specific parts of a computer system. One option for storage structures is to protect them with error correcting codes (ECC), as in IBM mainframes [66]. Combining ECC for arrays with DIVA avoids costly DIVA recoveries. However, ECC protection of arrays is on the critical path for array access (both read and write), and it will thus add to the microprocessor's critical path and degrade its performance in the fault-free case. Storage structures can also be protected by using a level of indirection to map out faulty portions of the structure. Whole disk failures were addressed by RAID [52]. For disk faults that did not incapacitate the entire disk, the solution was to map out faulty portions at the sector granularity. Similar approaches have been developed for DRAM main memory. Whole chip failures are tolerated by chipkill memory and RAID-M [22, 27], and partial failures are tolerated with schemes that map out faulty locations [19, 44, 59]. For SRAM caches, techniques have been developed to map out defective locations during fabrication [84] and, more recently, during execution [49].

3 Extending DRAM Use to the Level 1 Data Cache in Throughput-Oriented CMPs

We present work in this chapter that is motivated by the same basic trends that we outlined in Chapter 1, but here we shift our focus from single-threaded performance-oriented cores to a throughput-oriented paradigm. With throughput-oriented workloads, the latency demands of individual threads are relaxed due to longer-latency events dominating end-to-end application response time. With subsequent CMOS process generations, microarchitects are afforded more transistors to work with, but physical packaging limits continue to stifle the growth of the power and cooling budget for a chip package.

The shift to chip multiprocessors (CMPs) has been fueled by these trends. While it is common to find a small number of concurrent threads to run on a CMP in most server and desktop settings, exploitation of tens to hundreds to thousands of schedulable contexts is limited in the general case. Fortunately, the Internet has brought with it a group of applications that commonly scale to tens of thousands of concurrent threads for popular sites. Web serving and web commerce middleware present the throughput-oriented CMP architect with a strong motivation to optimize a design for power-efficient throughput.

Throughput-intensive computing is not a new concept in the architecture community. Processors such as Piranha [9] and Niagara [62] have designs that favor throughput over single-thread latency reduction. The realization that power will constrain future CMP designs more than transistor budgets has led to a series of studies of what the optimal core design is for a throughput-intensive CMP. Li et al. [35] and Davis et al. [21] both survey the design space for cores by looking at in-order and out-of-order scalar and superscalar core architectures. They sweep the design space of the cache hierarchy as well by varying capacity and latency based upon the core that the cache is paired with. These studies started with a supposition that low latency would provide best perfor-

mance for the core. Cache designs that were paired with the various cores studied were chosen with low-latency favored. The work in this chapter is motivated by the observation that this focus on low-latency has resulted in the architecture community overlooking better-suited cache alternatives in deep submicron CMOS technology generations. Our hypothesis is that shifting from a low-latency cache design to one that more closely matches throughput-oriented core demands will create opportunity to trade latency for power savings or additional capacity, both of which are more beneficial to the throughput-oriented core.

In Section 3.1 we begin with an overview of the experimental methodology we employ for this work. Our research is then presented as a three-step progression to show that a DRAM-based L1 data cache is well-suited to a throughput-oriented core and workload. The three pieces of research that bring us to our conclusion are presented as follows:

1) First, we seek to understand the demands of throughput-oriented workloads on the on-chip cache hierarchy. In Section 3.2, we study two common throughput-oriented workloads to understand their demands.

2) After understanding the demands of the core and its workload, we then must look at what alternatives exist in the cache implementation. In Section 3.3 we present alternative cache designs to match the implementation of the throughput-oriented core to its cache hierarchy.

3) Finally, we present an evaluation of an L1 data cache, based upon DRAM storage cells, in a throughput-oriented system. In Section 3.4, we compare a throughput-oriented cache implementation to existing and possible proposed alternatives. In our evaluation, we show that our throughput-oriented L1 cache, matched to our core demands, provides better power-performance than existing or proposed alternatives. We also show the opportunity presented in shift-

ing the L2 cache from a traditional SRAM-based implementation to a DRAM-based implementation.

We conclude the chapter with a review of related work in Section 3.5.

3.1 Experimental Methodology

In contrast to the research conducted on fault-tolerance in Chapter 2, our work here has a scope that is beyond the single microprocessor core. Since we are evaluating cache architectures for throughput-oriented CMP core designs, we require a system-level simulation capability that includes a detailed model of the core and on-chip cache hierarchy.

We examine the throughput-intensive workloads' memory demands with Simics [42] using the GEMS [43] detailed processor and memory simulation modules (Opal and Ruby), modified to allow us to simulate simple in-order and out-of-order cores. We utilize a base, 2-wide core design, evaluating both in-order and out-of-order versions of this base. Details of the simulated machine configurations can be found in Table 3-1. For all simulations performed with these workloads, we run a fixed unit of work, to measure the application-level throughput improvements from the configuration changes we apply.

For purposes of normalized performance comparison, we utilize a baseline core design that supports two simultaneous threads. We focus our study on 8-threaded and 16-threaded core configurations, based upon the previously published results on throughput-oriented core design [21, 35]. Early experiments we performed to assess the design space corroborated the results from these previous studies and showed no advantage to the 4-threaded core configuration, so we discarded it from further consideration.

Table 3-1. Processor Configuration for L1 Data Cache Latency Sensitivity Study

Property	Configuration
Core Configuration	In-Order/Out-of-Order, SPARC III+, 7-Stage Pipelined
Width (Fetch/Decode/Issue/Execute/Commit)	2/2/2/2/2
Scheduling/Instruction Window Size	4/16
L1 Instruction Cache Size/Latency/Associativity	16KB/1 Cycle/4-Way Set-Associative, 64B Lines
L1 Data Cache Size/Latency/Associativity	16KB/1-8 Cycles/4-Way Set-Associative, 64B Lines
L2 Unified Cache Size/Latency/Associativity	16MB/12 Cycles/4-Way Set-Associative
Main Memory Latency	160 Cycles
Branch Predictor	YAGS
SMT Support	2, 8, or 16 Threads
SMT Fetch Policy	1 Thread/ Cycle, Lowest Retired Count First
Operating System	OpenSolaris Nevada Build 87
Workloads (Units of Work)	Apache 2.2.9 (1,000 web transactions), SPECjbb 2000 (10,000 transactions)

We use Cacti 5.3 [74] to model the caches with capacity from 16 KB to 1 MB, from 1 to 64 banks, and with different storage cell technologies. Details of our cache configurations are shown in Table 3-2. We compare caches composed entirely of SRAM (traditional designs), those that use DRAM as the data and tag storage array element, and those that use a hybrid SRAM/DRAM storage cell with SRAM tags [77]. We modified Cacti to model the implementation of both tags and data in the DRAM process.

We estimate hybrid cell values from Cacti data for SRAM and DRAM combined with data from [77]. Specifically, for power, we use SRAM values for dynamic energy per read or write. This is because more than 95% of accesses hit in the SRAM way of the hybrid cache. We factor a 75% reduction in bank leakage power over SRAM, as the authors forecast. We derive rough area estimates for the hybrid cell-based cache by scaling the cell-technology independent components

Table 3-2. L1 Data Cache Configurations Explored

Parameter	Value
Technology	32nm
Cache Size	16KB, 32KB, 64KB, 128KB, 256KB, 512KB, or1MB
Width	64-Bit Data, 64B Blocks
Storage and Tag Cells	All SRAM or All 1T1C DRAM or Hybrid [77]
Optimization Target	Energy-Delay
Banks[a]	1, 2, 4, 8, 16, 32, or 64
Architecture	Uniform, Non-Blocking Cache Architecture with Pipelining
Port Configuration	2 (1 Read/Write, 1 Single-Ended Read)
Associativity	4-Way, Set-Associative

a. Cacti enforces a minimum bank size of 32 sets, limiting some configurations.

of the area data Cacti provides and factoring a 50% reduction in area of the data array over that of an equivalent 6T SRAM cell array, as the authors indicate.

Additional performance factors must be considered when hybrid cells are in use. The hybrid cell may incur additional L1 to L2 traffic due to its writeback policies to avoid having to refresh the DRAM part of the cell. This will be workload dependent, so we assume a best case workload that incurs no additional writeback traffic. We also do not factor the additional circuitry for managing the swapping of data from the DRAM storage to the SRAM storage in the cell nor do we factor the canary cell for managing early writeback. Finally, we assume that all hits in the hybrid cache can be serviced in 1 cycle (DRAM hits take 3 cycles in the proposed implementation). We factor the extra reads and writes that a swap incurs and assume that 5% of the accesses to the cache incur a swap, based upon data from [77] on 16 and 32KB, 4-way set-associative hybrid caches running SPEC benchmarks.

3.2 Demands of Throughput-Oriented Workloads

In order to develop a throughput-oriented cache design, we first must understand the demands of the throughput-oriented core, running representative applications, on the cache hierarchy. In this section, we determine the demands of the throughput-oriented core on the cache. We leverage past work on throughput-oriented core design, which indicates that a relatively simple, narrow core with SMT support is a power-efficient throughput engine. We first address how much bandwidth is demanded by such a core (Section 3.2.1). We then explore whether the core's cache demands are sensitive to L1 data cache latency (Section 3.2.2).

3.2.1 Bandwidth Demands of the Throughput-Oriented Core

Our desire is to balance the core's demands with the cache's supply to optimize throughput. To that end, we need to understand the memory bandwidth demands of a throughput-oriented workload, then map it to our core building block to arrive at an upper bound for what the core will require. In order to maximize the throughput of a highly-threaded CMP, we need the cache subsystem to meet the bandwidth demands of the workload.

Our bandwidth study consists of an experimental evaluation of a representative throughput-oriented core design. Memory operation mixes for different thread-counts on the core for each benchmark only differ in a statistically significant way for the 16-thread SPECjbb configuration, where we see almost 2% more loads in the dynamic instruction stream over its 2-threaded and 8-threaded counterparts. Apache has ~20% loads and ~9% stores in the dynamic instruction stream. SPECjbb has ~17% loads and ~7% stores. This data is detailed in Figure 3-1. Apache's greater proportion of memory instructions should result in a higher overall bandwidth demand. Averages were taken over tens of runs for each thread configuration shown. Error bars show one standard deviation from the averages.

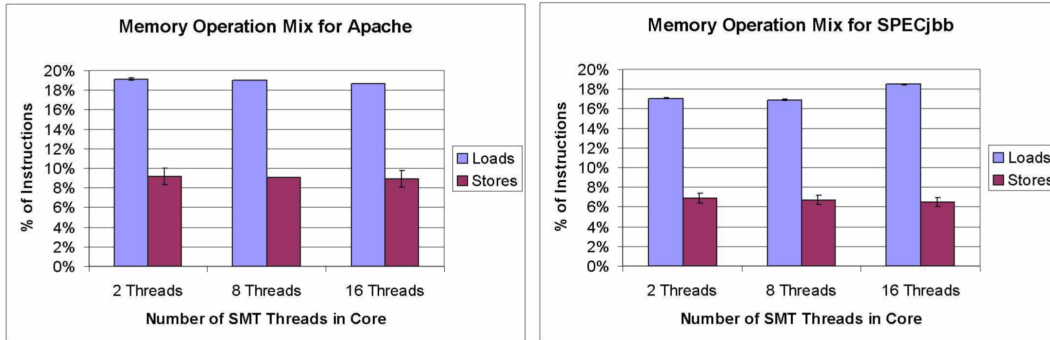


Figure 3-1. Dynamic Instruction Stream Memory Instruction Mix

Data collected shows similarity across core thread configurations. We observe slightly greater memory instruction percentage in Apache with behavior for both benchmarks within established norms of roughly 33% memory instructions with a 2:1 load:store ratio within that 33%.

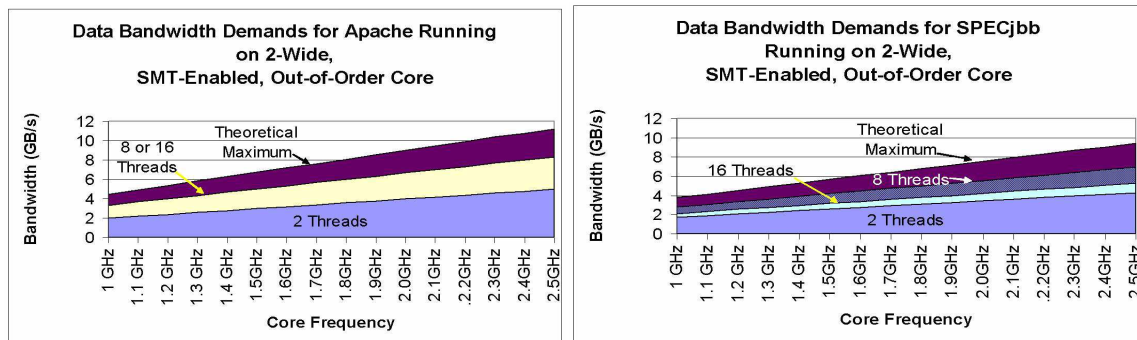


Figure 3-2. Throughput-Oriented Core Bandwidth Demands on L1 Data Cache

Results show actual measured bandwidth consumed on throughput-oriented 2-wide superscalar core as well as theoretical maximum if core IPC is equal to core width of 2.

We show bandwidth demands for core operating frequencies between 1 and 2.5 GHz in Figure 3-2. The maximum theoretical demand is derived by assuming perfect utilization on our core (i.e., IPC=2) multiplied by the measured memory instruction mix and core operating frequency. This represents an upper bound for these workloads on a 2-wide superscalar core. At 2.5 GHz, this upper bound is just over 11 GB/s for Apache and just under 10 GB/s for SPECjbb. Actual bandwidth requirements were calculated by taking the actual IPC instead of ideal IPC for the maximum theoretical limit. We found in our simulations that bandwidth demands were just over 8 GB/s for Apache and just over 7 GB/s for SPECjbb. The disparity here represents the gap between the actual IPC of the core during our experiments and the theoretical maximum IPC of 2.

From this study, we conclude that the throughput-oriented core bandwidth demands on the L1 are met by a wide variety of cache implementations. The use of a simple, low-power core and the typical instruction mix of the workloads we target are the essential elements that lead to the relatively low bandwidth demands on the L1. We do note that meeting the bandwidth demands of a core running at the top of the frequency range we examine will require an average of more than one 8-byte datum per cycle to be served from the cache to the core. This leads us to focus on 2-port caches with the ability to issue two reads or one read and one write per cache cycle in Section 3.4.1.

3.2.2 L1 Data Cache Latency Sensitivity

The second aspect of cache demand, in addition to bandwidth, is whether the core is sensitive to cache latency. While evidence exists in the prior design space studies [21, 35] to support the intuition that throughput-oriented cores may not need low-latency caches to perform well, the design space is sparsely explored. In order to better understand the sensitivity of representative Internet workloads to first-level cache latency, we performed a simple limit study. In this set of experiments, we ran Simics with Opal and Ruby GEMS modules configured as originally shown in Table 3-1. The goal here is to determine if we can mask latency effectively by increasing the number of threads.

The results of our study are shown in Figure 3-3 and Figure 3-4. We use a 2-threaded core as our base configuration for comparison purposes and show results for the 8-thread and 16-thread core configurations. For both in-order and out-of-order core configurations, we observe improved throughput when we have eight threads, even though we hold other core resources fixed. The improvement in throughput by adding additional threads confirms our hypothesis that additional thread contexts can mask data cache access latencies as well as other stall-inducing events in indi-

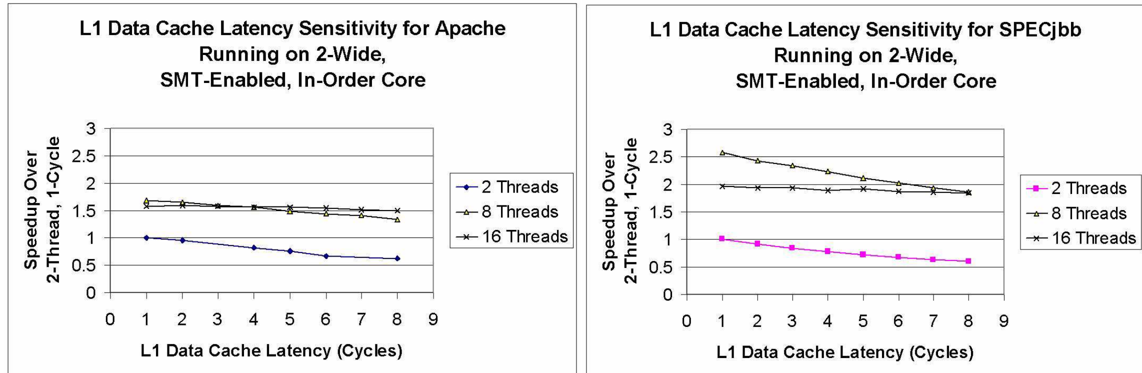


Figure 3-3. Apache Web Server and SPECjbb Normalized Throughput on Out-of-Order, Multithreaded Cores with Varying L1 Data Cache Latency

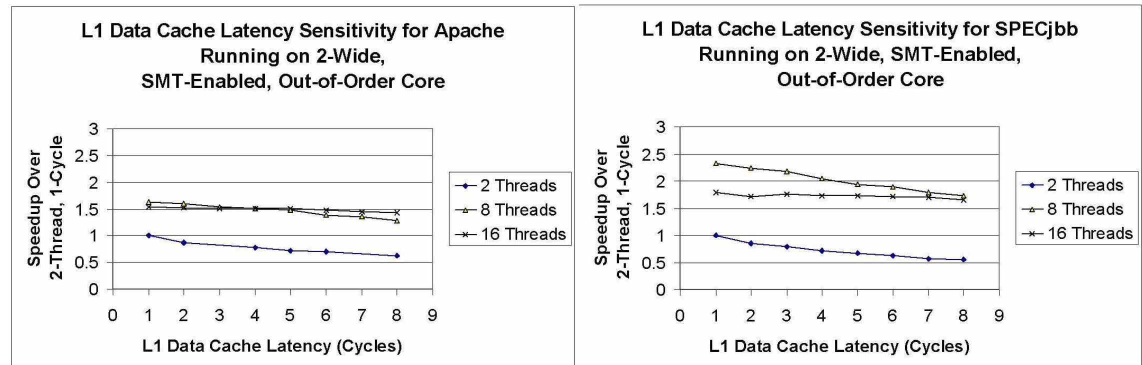


Figure 3-4. Apache Web Server and SPECjbb Normalized Throughput on In-Order, Multithreaded Cores with Varying L1 Data Cache Latency

vidual threads' dynamic execution. The reduction in speedup as we increase L1 data cache access latency is evidence that we cannot fully mask latency effects with this technique. We observe no loss of speedup over the range of L1 data cache latencies we studied with the 16-thread core configuration. However, at low latencies, the 16-thread core under-performs the 8-thread core and only reaches parity performance with its 8-thread counterpart at 8-cycle L1 data cache latency for SPECjbb. We conclude that the 16-thread core is overthreaded.

There are multiple potential bottlenecks within the core that lead to this overthreaded situation. The use of small instruction windows and issue queues as part of our low-power core model limits the ability of out-of-order execution to extract instruction-level-parallelism (ILP) from the individual threads running on the core. This effect is compounded as we increase the number of simulta-

neously-active threads. In cases where we have 8 or 16 threads, the in-order core effectively has 8 or 16 instructions, respectively, ready or executing in a given cycle. With the size of window we use, that means that the number of additional instructions that out-of-order execution makes possibly executable is 0 or 8 for these highly-threaded cores. For this reason, we see little difference between in-order and out-of-order performance, and the level of threading is the primary driver of the observed effects.

3.2.3 Throughput-Oriented Cache Demand Summary

From this study, we arrive at a compound answer to the question of what the core demands are on the cache. For bandwidth, we need a maximum of 12 GB/s and we can achieve additional throughput with extra threads, thus effectively overcoming performance losses due to L1 data cache latency at lower numbers of threads per core. We will use this fact in Section 3.4.3.1 to show how this latency tolerance can be converted to additional throughput by tailoring of the L1 data cache design. This correlates with results from previous core design studies [21, 35] and it is the design point that we focus our efforts on meeting the demands of throughout the rest of this chapter.

3.3 Cache Building Block Technology Alternatives

Now that we understand the basic demands of our core, we next want to examine alternatives to meeting those needs in our cache design. In this section, we are seeking the building block that provides a resulting cache that best matches our core demands to provide optimum throughput under a fixed power budget. To get to that cache design, we will first dive into technology alternatives we can employ as the building block base for our cache. We review alternative memory cell technologies that have been proposed first. Some of these are already in use in applications inside or outside of the L1 cache. We compare and contrast the primary operating characteristics of these

building blocks before evaluating their performance in the data cache, starting with the L1 data cache in Section 3.4.1 and concluding with the L2 cache in Section 3.4.2.

As we move into deep submicron CMOS processes, leakage effects on a traditional 6T SRAM cell have inspired research into alternative cache implementation technologies. Here, we review the techniques most germane to a throughput-oriented design. These fall into three major categories - improving SRAM-based implementations by reducing their susceptibility to leakage effects, replacing SRAM with embedded DRAM, or utilizing a hybrid approach that combines SRAM and DRAM to meet the desired characteristics of the cache that the cell is used in.

To understand what cache technology best meets throughput-oriented core demands, we need to understand the strengths and weaknesses of these three option categories and then employ the most appropriate technology for the design. In the next three sub-sections, we will discuss the alternatives and their general properties before we propose a throughput-oriented design and move into a more detailed comparison of caches composed of these three types of cell.

3.3.1 Build a Better SRAM Cell

In complement to materials efforts, new SRAM cell designs have been proposed in order to limit the static power dissipation of SRAM-based caches at sub-threshold voltages. Techniques proposed include asymmetric designs [7], use of low-voltage standby modes [50, 54], and deactivation of SRAM cells that are unused or not likely to be used soon [32, 33]. These improve cell leakage characteristics, but do not eliminate them.

A number of novel SRAM cell designs that utilize additional transistors have been proposed to support stable operation at lower operating voltages. Examples of these techniques include 8T [17], 9T [39], and 10T [15] cell designs. Each of these trades additional die area and active power

for better performance and/or leakage characteristics. Latency for these designs is similar to standard 6T cell design latencies.

3.3.2 Embedded DRAM as an Alternative to SRAM

With its low static power consumption and higher density, DRAM has emerged as an on-chip alternative to SRAM for the last-level caches on the chip. Initial implementations [29] have utilized a 1T1C cell design, typical of off-chip memories. The relatively long cycle time of 1T1C DRAM memories, due in part to destructive reads, has prevented their use closer to the processor core. We focus on the use of 1T1C DRAM cells and will show that the latency drawbacks they bring to the design can be overcome for a throughput-oriented application.

Luk et al. [41] proposed a 3T1D DRAM memory cell design with latency comparable to 6T SRAM and non-destructive reads. Liang et al. [37] propose use of this cell for a register file application and show how a short refresh time allows the latency of the design to match its SRAM counterpart with better process variation tolerance and comparable power characteristics. Liang et al. also explore use of this cell in future CMOS generations as a replacement for the 6T SRAM cell [36, 38]. These efforts seek to maintain parity performance characteristics of SRAM with an alternative cell design that provides better process variability tolerance. Our use of a 1T1C DRAM cell also evades the issues with the 6T cell's sensitivity to process variation, and we show that, for a fixed cache power budget, a cache composed of 1T1C DRAM is a better throughput booster for a throughput-intensive workload than a lower-latency SRAM or DRAM alternative. In our work, we do assume that 3T1D cell would be used in the L1 instruction cache and the register file, where latency is critical to throughput.

3.3.3 Hybrid Caches

A novel hybrid cache design has been proposed to bring eDRAM to the L1 cache [77]. In this design, the first way of the L1 is implemented with SRAM, with subsequent ways of an n-way set-

associative L1 cache being implemented with eDRAM. A most-recently used (MRU) heuristic is developed to optimize the hits to the SRAM way by moving data dynamically based upon predicted hit patterns. Analysis of the MRU algorithm with single-threaded SPEC benchmarks on a fast, wide, out-of-order core show that less than 5% of the hits to the L1 fall in the slow ways.

The goal of the hybrid implementation is to provide nearly the energy-efficiency of eDRAM with nearly the low latency of SRAM. In effect, this work provides a two-part L1 cache. The first way is the fast-hit way, and efforts are made to ensure that the most-likely-to-be-accessed data is kept in this way (at the expense of extra power and logic to heuristically migrate data from the other ways). The remaining ways are slower to access, but faster than the L2. We argue that the effort to retain high-frequency, low-latency operation is unneeded and show that this technique, at best, provides parity performance in a throughput-oriented application at a higher power cost.

3.4 Experimental Cache Design Space Exploration

We propose using DRAM instead of SRAM to arrive at a throughput-optimized L1-data and L2 cache design. We hypothesize that, of the available cell technology alternatives, use of DRAM will provide the greatest opportunity to improve throughput under a fixed power budget. We now test this hypothesis experimentally for the L1-data and L2 cache. For our study, we select the 6T SRAM cell as our basis of comparison, as it is pervasive in its use in the L1 data cache today and widely used in the L2 as well. For alternative storage cell technologies in the L1 data cache, we use 1T1C DRAM and the hybrid cache proposed in [77]. With these three alternatives, we will model typical cache capacities in each technology and compare bandwidth, power, area, and latency characteristics to find the best match for our core in Section 3.4.1. We then compare 6T SRAM and 1T1C DRAM for the L2 cache in Section 3.4.2 and conclude in Section 3.4.3 with a

summary of our results and the unified cache design proposal that our experimental evaluation arrives at for a throughput-oriented design.

3.4.1 L1 Data Cache Evaluation

Our first set of experiments seeks to examine the alternatives for a throughput-oriented L1 data cache. We choose capacities in a range from present L1 data cache norms to the point at which the capacity of the cache would likely overwhelm the L2 cache capacity in an inclusive cache hierarchy. The goal of this evaluation is to find the best match for the bandwidth demands of our core as a segue to exploring how a better-matched cache can improve core throughput.

For each of the cache capacities and technologies we compare, we simulated all possible bank configurations up to 64 banks. Cacti returned energy-delay optimal configurations for each valid configuration. From the set of results at a given technology and capacity node, we selected the cache design that provided the lowest energy cost per unit of bandwidth at an operating frequency of 1.5GHz[1].

3.4.1.1 Bandwidth and Power

Our intuition is that bandwidth and power will be the two most critical factors to match to the core demands. Because we can easily trade bandwidth for power if we run the cache at a lower clock rate, we consider these two metrics together in our discussion. Figure 3-5 shows the cache bandwidth available to a core operating at frequencies depicted on the x-axis. We show the maximum theoretical bandwidth demands of the workloads from our study in Section 3.2.2 with dashed lines for reference. Because all of our selected cache configurations, regardless of technology of the storage cell, have at least two ports and at least two banks, the solid lines reflect the bandwidth

1. The 1MB DRAM design is only capable of sustaining a 1.2GHz maximum clock, so all data is shown relative to that reduced maximum clock rate, and labeled explicitly. While other designs in 1MB DRAM space meet the speed criteria, they come with increased power per unit throughput, so were not selected.

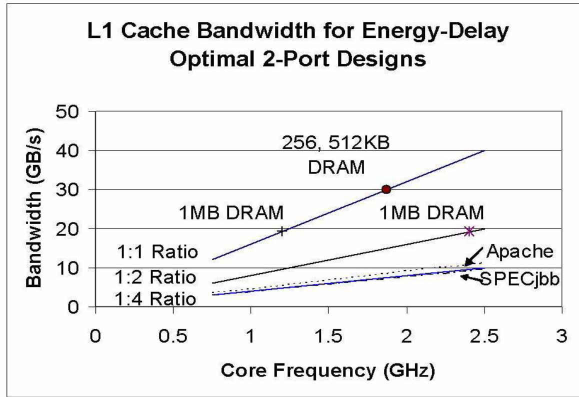


Figure 3-5. L1 Data Cache Bandwidth for Selected Designs as Core Frequency is Varied
Bank cycle time limits larger DRAM configurations to a lower maximum frequency.

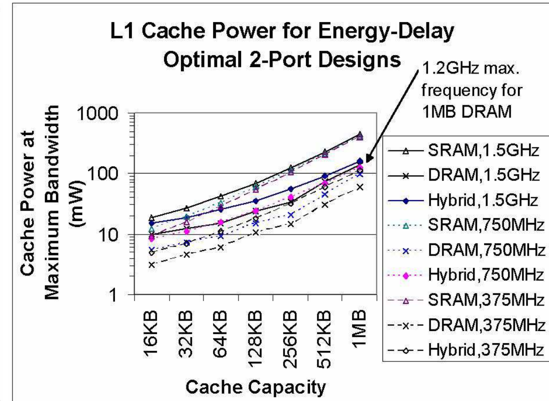


Figure 3-6. Power Comparison of SRAM, 1T1C DRAM, and Hybrid [77] L1 Data Cache Designs

available from the cache to a core at core:cache clock ratios of 1:1, 1:2, and 1:4. For 1T1C DRAM-cell based configurations, we cannot support 1:1 and 1:2 core:cache clock ratios for all frequencies in the displayed range. The frequencies at which a particular DRAM design is limited on a given line are shown and labeled explicitly in the figure. Specifically, with DRAM, we observe that the bank cycle time limits us to an achievable frequency of 1.2 GHz for our selected 1 MB cache designs and 1.87 GHz for 256 KB and 512 KB cache designs. This maximum frequency increases if we shrink the cache size or select a less energy-efficient design.

We based our bandwidth figures on a hypothetical maximum achieved when one bank per port was active every cycle. In designs with an equal number of banks and ports, it is assumed that all available banks are used every cycle to arrive at the maximum theoretical bandwidth value.

The other element of bandwidth is operating frequency of the cache. This is limited by the bank access time, which cannot be pipelined, and therefore becomes the limiting factor for increasing the cache frequency. For larger DRAM designs, this is a direct limitation in the operating frequency range of the core that we study. For SRAM designs, we are not constrained by bank cycle time in any of the studied designs. In the case of the hybrid cell, we believe that bank cycle times

would be faster than DRAM-only alternatives, since the hybrid cell allows reads to be destructive, avoiding the additional time to refresh the cell data after the read. We assume this would allow the hybrid design to equal SRAM in the core frequency range that we focus on.

From our previous study of core demands, we find that we have ample bandwidth available, even when we are unable to run the cache at the core frequency. We therefore model running the cache at a clock rate less than the core in order to reduce its power footprint. The result of slowing the clock is a net reduction in maximum dynamic power due to a reduction in bandwidth, since we limit the number of memory operations per second that the cache can support. We model a simple clock slowing solution that utilizes the core clock, creating a virtual cache clock that transitions on every *nth* edge where *n* is a factor of 2. We also assume that static (leakage) power remains fixed, since we do not assume a reduction in operating voltage with the speed reduction.

In our sweep of the design space, we explored core to cache clock ratios of up to 1:4, since we show that we cannot provide sufficient bandwidth to meet core demands at speeds slower than 25% of core frequency. In Figure 3-6, we show the power dissipation for caches implemented in the three technologies. For each base storage cell technology alternative, we plot three curves, one for each of the core:cache ratios we show in our bandwidth figure. We set the core frequency to 1.5 GHz, which represents a median value for where we forecast throughput-oriented CMPs to clock cores at. We cap maximum frequency to avoid distortion of the SRAM and hybrid values from the additional active power from operating at a higher frequency. That is, the SRAM and hybrid caches can run at a higher clock rate to provide greater bandwidth, and therefore, greater active power.

At small cache capacities, the absolute power numbers for all caches are small - less than 19 mW for the 6T SRAM 16 KB design running at 1.5 GHz. Despite the low absolute power values,

the clear power advantage of hybrid and DRAM-based cache designs is evident. Their 16 KB-capacity cache power comes in at roughly 15 mW and 10 mW, respectively. These gains are due to the reduced leakage power component for the DRAM and hybrid designs. As we scale to larger capacities, this advantage is amplified as data and tag cells become the dominant contributors to overall cache power footprint.

Scaling cache operating frequency to reduce dynamic power is effective, netting additional gains that are relatively constant across the sweep of cache sizes. The uniformity of power reduction effect across technologies with cache frequency reduction is to be expected, as the bandwidth of all of these caches is independent of their capacity and is constant at a given cache frequency[2]. We show the 1:2 core:cache clock ratio in Figure 3-6 with a fine-dashed line and the 1:4 ratio with a coarse-dashed line. The series markers are matched to the cell-basis of the cache implementation to help the reader discern the important relationships between the three cache bases we study. Note that the three SRAM curves appear to converge as cache size increases. This is due to the leakage power domination of the total power of the SRAM cache. The reduced leakage power of the alternative-cell based caches makes reducing the operating frequency of the cache to reduce total cache power more effective.

Our examination of this data leads us to two conclusions. First, all of our cache designs provide ample bandwidth to satisfy the demands of the core when clocked at the core frequency. Even when we move to a 1:4 core:cache clock ratio, we still see sufficient bandwidth to support almost all of the maximum theoretical bandwidth that our workloads will demand on our core. Second, DRAM and hybrid cell-based caches have a distinct power advantage over their SRAM counter-

---

2. The exception in the presented data is the 1 MB DRAM cache solid-curve (1:1 core:cache clock ratio) data point, which reflects its maximum operating frequency of 1.2 GHz. It is explicitly labeled in the figure.

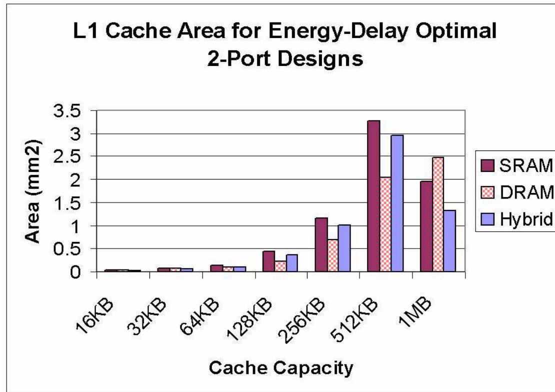


Figure 3-7. L1 Data Cache Area for Energy-Delay Optimized Designs Implemented with SRAM, DRAM, or Hybrid Cells

Trend discrepancies in 512KB and 1MB areas are due to differing bank configurations.

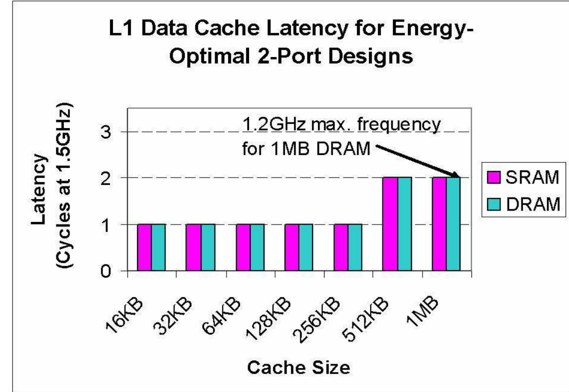


Figure 3-8. Latency of SRAM and DRAM-Based L1 Data Cache Implementations

Note equivalence of these technologies at a target core frequency of 1.5 GHz. Hybrid cells have variable latency, so are not shown.

parts, but this advantage is small, in absolute terms, at the relatively small cache capacities typical of an L1.

3.4.1.2 Cache Area

At the typical capacities of L1 caches that we see today, cache area is not generally seen as a limiting factor. We included it in our study since we do examine larger-than-typical cache capacities, which start to see non-trivial area differences between various designs.

We show the areas of the selected cache configurations in Figure 3-7. Our optimization on energy-delay efficiency results in Cacti converging on different bank configurations when we move from SRAM or hybrid cells in the data array to DRAM at sizes of 512 KB and 1 MB. The result is 512 KB designs that have more banks than their 1MB counterparts in the SRAM and hybrid cell implementations. This causes the 512 KB caches to be less area-efficient, which makes them significantly larger. At 1 MB capacity, the SRAM and hybrid-cell based caches have 8 banks, while the DRAM-based cache has 16, leading to the non-intuitive larger size of the DRAM-based cache. We note that areas for caches smaller than 128 KB are dominated by port and wire layout,

leading to very little overall difference in footprint when moving between these capacities. Starting at 128 KB, storage cell area begins to dominate, at which point we see a marked advantage for 1T1C DRAM over the other two alternatives for configurations where we have identical numbers of banks. Since we assume area is at less of a premium than power in our study, we are willing to select area-inefficient configurations when they provide lower power per unit bandwidth.

3.4.1.3 Cache Latency

Latency is important when we compare designs; while we can tolerate latency in a throughput-oriented core, we do observe better throughput at lower latencies, so, all other factors being equal, we would prefer low-latency operation. When we examine latency characteristics of the selected cache designs, we see relative homogeneity of access latency at the frequencies and capacities we target. Latency data in Figure 3-8 shows that sizes above 512 KB require a single pipeline stage be placed into the cache in order to sustain a 1.5 GHz[3] cache clock. For all of the designs we present, this single pipeline stage would be sufficient to sustain up to the 2.5 GHz clock rate that we use as an upper bound for core frequency. Further, none of the configurations with single-cycle latency at 1.5 GHz would require an additional cycle of latency at 2.5 GHz.

The latency of the hybrid cache is variable, by design. In the proposed implementation [77], the authors indicate that tags and way 0 are accessed in parallel and that hits to the DRAM ways are serviced only after a tag hit is verified. This causes DRAM-way hits to have a latency that is longer than their counterparts in an all-DRAM cache, but this effect is mitigated by the relatively low fraction of hits in the DRAM ways. While they might not reach the same speed as their SRAM counterparts, we optimistically assume that the hybrid cell designs will have latency characteristics that match the SRAM and DRAM data when we do throughput improvement comparisons.

---

3. Again, the exception is the 1MB DRAM cache, which can only run at a maximum frequency of 1.2 GHz, and requires a pipeline stage at that frequency.

3.4.1.4 Summary of L1 Data Cache Evaluation

The point of this study was to answer find an L1 data cache that best fits throughput-oriented core demands. Our finding that all of our caches provide more than enough bandwidth at their maximum operating frequency leads us to conclude that a balanced implementation may intentionally lower the frequency of the L1 data cache, introducing latency, reducing superfluous bandwidth and dynamic power in the process. We now turn to the question of how we can gain additional chip throughput with a cache better matched to our core. We only want to constrain the design if it results in an opportunity for throughput improvement. That is, we do not want to constrain the cache if doing so nets no additional throughput benefit to the overall CMP.

The most obvious benefit we get in exchange for capping our bandwidth is a reduction in power. Moving from SRAM to DRAM gives us the best power savings. Slowing the L1 data cache down by reducing its operating frequency and introducing additional latency amplifies this effect, while keeping us within our core's bandwidth demands. Unfortunately, the absolute numbers we find for typical L1 data cache sizes do not give us a large amount of power to work with if we seek to employ the savings we gain from moving to DRAM. An alternative to reallocating the power savings to another component is to expand the size of the L1 data cache. Our shift in technology puts us on a different power curve. This leap allows us to increase capacity in the cache. The benefits of this trade-off are something that we explore in Section 3.4.3.

3.4.2 L2 Cache Evaluation

For the L2 cache, the argument for use of DRAM is compelled equally by density as it is by power savings. L2 caches on throughput-oriented CMPs scale as the number of cores (and L1s) that they have to support. This scaling quickly becomes problematic with SRAM in a 32 nm process. When we simulate 16 MB, 32 MB, and 64 MB L2 cache configurations, we see the chal-

Table 3-3. L2 Cache Configurations Explored

Parameter	Value
Technology	32nm
Cache Size	16MB, 32MB, and 64MB
Width	256 Bits, 256B Blocks
Storage and Tag Cells	All SRAM or All 1T1C DRAM
Optimization Target	Energy-Delay
Banks	1, 2, 4, 8, 16, 32, or 64
Architecture	Uniform, Non-Blocking Cache Architecture with Pipelining
Port Configuration	12 (4 Exclusive Write, 8 Single-Ended Read)
Associativity	4, 8, and 32-Way, Set-Associative

lenges facing the throughput-oriented architect. The details of the L2 cache configurations we simulated are shown in Table 3-3. We did not include an evaluation of hybrid cell-based L2 caches since the hybrid cell design requires an on-chip backing store for the dynamic ways, which are not refreshed (we do not assume an L3 cache). The results here also do not factor any on-chip interconnection network nor do they include modeling of contention for ports, which is workload and chip-design dependent. The use of NUCA cache architectures here may be warranted if latency or maximum operating frequency are found to be critical factors in a design. L2 interaction with on-chip networks is outside of the scope of this work.

We see in Figure 3-9 that a 16 MB SRAM cache has an area of ~120 mm^2 - a large portion of available die capacity - with DRAM coming in an order of magnitude smaller at ~12 mm^2. Figure 3-10 shows that this reduction in size does enable faster access, but the properties of DRAM again keep cycle times (and maximum clock rates) limited. We found maximum clock rates for SRAM in the 2.5 GHz range, while the DRAM caches were limited to between 980 MHz and 1.33 GHz. The SRAM designs require twice the pipelining of their DRAM counterparts, again due to the order of magnitude difference in the size of the cache. Here, the density advantage of

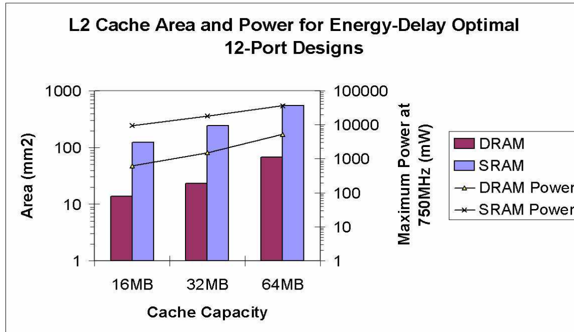


Figure 3-9. Area and Power Characteristics of Large L2 Cache Configurations for Large-Scale CMPs

Note that 32 MB and 64 MB capacities would not fit on chip in 32nm process with SRAM.

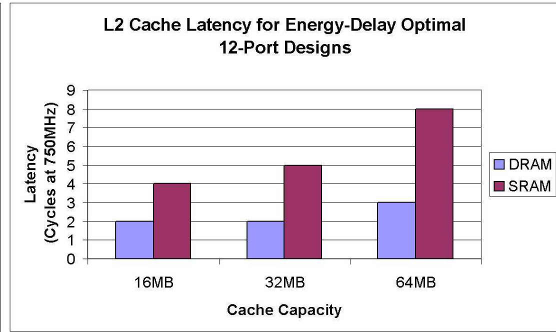


Figure 3-10. Latency to Access SRAM and DRAM L2 Caches at 750MHz Operating Frequency

DRAM advantage is due to smaller area, which results in shorter wires.

DRAM, which is the source of the order-of-magnitude area reduction also results in less wire delay from the port to the banks in the cache. So, the bank cycle time limits DRAM to lower frequencies, but the density of DRAM provides area and latency improvements over SRAM.

The most important result here is the power footprint of a DRAM-based L2 cache. A 16 MB L2 SRAM-based cache consumes ~10 W, or roughly 5-10% of the chip's power budget. If we switch to DRAM-based cache, we reduce power to less than 700 mW. Moving to cache capacities that are unrealizable with SRAM, we still have power footprints less than the 16 MB SRAM. A 64 MB DRAM L2 at 750 MHz operating frequency consumes just under 6 W of power at maximum bandwidth.

3.4.3 Putting it All Together: Evaluation of a Throughput-Oriented Cache

We have shown that running the L1 data cache at a lower frequency than the core will provide additional power savings with a limited impact on core throughput. Note that running the cache at a reduced clock rate both increases latency and reduces bandwidth. If we do run the cache at a reduced frequency, data in Figure 3-6 suggests that the power savings in the L1 data cache are not sufficient to amortize the power costs of an additional core on the die. Even if we assume the max-

Table 3-4. Cache Power, in Milliwatts at Maximum Throughput[a]

Type/Capacity	16 KB	32 KB	64 KB	128 KB	256 KB	512 KB	1 MB
SRAM @ 1.5 GHz	**18.6**	26.6	41.8	69.5	125.6	231.6	437.7
DRAM @ 1.5 GHz	10.0	12.3	**15.0**	23.5	34.1	74.4	141.2
Hybrid @ 1.5 GHz	**15.2**	18.8	25.5	35.3	55.9	90.4	158.6
SRAM @ 750 MHz	**12.4**	19.4	32.9	59.0	111.2	212.4	408.7
DRAM @ 750 MHz	5.4	7.1	9.1	**14.8**	21.1	45.1	96.5
Hybrid @ 750 MHz	8.4	10.9	**15.7**	23.7	40.1	69.4	126.8
SRAM @ 375 MHz	9.3	**15.9**	28.5	53.7	103.9	202.9	394.3
DRAM @ 375 MHz	3.1	4.6	6.1	10.5	**14.6**	30.4	59.2
Hybrid @ 375 MHz	5.0	7.0	10.9	**17.9**	32.1	58.9	110.9

a. Highlighted cells show configurations that fit within the budget of a 16 KB SRAM cache at 1.5 GHz.

imum power savings of roughly 15 mW per core, a 2 W core would only be amortized if we had 133 cores already on the die. At a roughly 100 W power budget for the package, this math does not work out in our favor. Our data does, however, indicate that we can allocate the power savings from this technique to larger cache capacity per core.

The next two sub-sections explore what we can do to improve CMP throughput if we match the cache to the core at both an L1 data and L2 level. In each section, we show possible tradeoffs that might be made and discuss their effects on throughput.

3.4.3.1 Improvements in the L1 Data Cache

In this study, we explored equal-power configurations that trade L1 data cache latency for capacity to see if further improvements in throughput can be gained. Since all of our modeled caches provide sufficient bandwidth to meet core needs, we include cache frequency reductions that trade both latency and bandwidth for additional power, which we put toward increasing capacity. We base our power budget on the 16 KB SRAM cache running at the core clock rate. We show a summary of cache power footprints in Table 3-4 to help clarify the equal-power alternatives. We

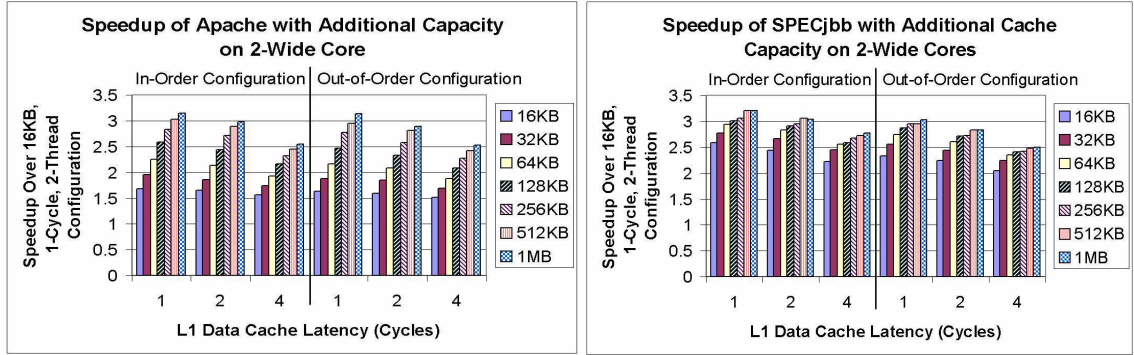


Figure 3-11. Speedup by 8-Thread Configurations Over 2-Thread Core with 16 KB, 1-Cycle L1 Data Cache at L1 Data Cache Capacities From 16KB to 1MB and 1, 2, and 4-Cycle Latencies

This figure shows the possible trade-offs that can be made in a design between adding additional capacity and increasing access latency.

highlight the largest-capacity alternatives that fit within the budget of a 16 KB SRAM-based cache running at 1.5 GHz in bold in each row of the table. For hybrid implementations, we can afford a 64 KB at a 1:2 core:cache clock ratio and 128 KB in capacity at 1:4 core:cache clock ratio. When we move from an SRAM based cache to a 1T1C DRAM-based cache, we can afford 64 KB, 128 KB and 256 KB capacities at 1:1, 1:2, and 1:4 core:cache clock ratios, respectively. The disparity between DRAM and the hybrid cell is because the hybrid cell only achieves, at best, 75% static energy reduction in a 4-way set-associative cache because it implements one way in SRAM. The dynamic power difference between SRAM and DRAM does not compensate for this with the bandwidth we achieve in our cache designs. The latency advantages of SRAM and the hybrid cell are of less use than the density and capacity advantages of DRAM toward our throughput-oriented design goals.

To illustrate the benefits of moving from SRAM to DRAM and then trading some latency for additional capacity, we show the speedup of the different core/cache/workload configurations in Figure 3-11. Each graph displays the speedup of an 8-threaded core's execution over its 2-threaded counterpart operating with a 16 KB L1 data cache with 1-cycle latency. If we select the largest-

capacity caches from Table 3-4 for our 18.6 mW power budget, we see the following effects. For SPECjbb, if we go from a 16 KB to a 64 KB L1 data cache capacity at the same latency, we get a throughput speedup of 14%, for in-order, and 18%, for out-of order cores. When we examine the longer-latency alternatives, either a 128 KB cache with a 2-cycle latency or a 256 KB cache with a 4-cycle latency, we see no additional speedup for the 128 KB cache and a reduction in throughput for the 256 KB, 4-cycle cache design. For Apache, the configuration that maximizes throughput is the 128 KB, 2-cycle latency cache. This nets a 45% and 43% improvement in speedup and for in-order and out-of-order configurations, respectively. The sweep of latencies and cache sizes shows that additional benefit (beyond that achieved with multithreading the core) is possible if we extend the cache capacity, even if it means a modest increase in latency to access the cache.

We do note that, while the bandwidth needs of the workload are most closely matched by the L1 clocked at a 1:4 core:cache ratio, the speedups gained are smaller. Here, the sensitivity of the core and workload to the extra latency of access outweighs benefits from additional capacity.

In this study, we assume that the cost of adding SMT support to the core has already been covered in the core's power budget. If that is not the case, it is clear that 8-thread SMT support is a first priority. Using Cacti to model the leakage power of an SRAM-based register file with 4 read and 2 write ports gives us an estimate of the magnitude of extra power cost that the core must carry to support multithreading. We do not scale the dynamic power of the register file in this estimate, since the width of the core remains fixed in an SMT configuration, limiting the bandwidth demands on the register file. Using this rough estimation technique, we find that roughly 40 mW of leakage power would need to be budgeted toward moving from 1 to 8 threaded SMT support in the register file.

3.4.3.2 Opportunities in the L2 Cache

In the L2, the magnitude of power savings when we move from SRAM to DRAM is significant. Here, we do have opportunity to trade cache capacity for additional cores. The gains in area-efficiency and power reduction (both static from cell leakage and dynamic from reduced wiring) result in a large enough savings to expect that the addition of one or more additional cores may be an option in the 32 nm process generation.

At a 750 MHz operating frequency, we find the DRAM L2 power to be 600 mW for the 16 MB configuration, 1.5 W for the 32 MB configuration, and 5.3 W for the 64 MB configuration. All of these caches also fit within a reasonable area on a 32 nm die, at 14, 32, and 70 mm^2, for the three sizes studied. The microarchitect has a choice for more capacity, more cores, or possibly more of another core resource that is shared amongst clusters of cores. Because the power and area savings here are orders of magnitude higher than in the L1 data cache, the possibilities expand. We leave detailed studies of possible tradeoffs in this space to future work.

3.5 Related Work

Our focus has been on finding a cache design that best meets the demands of a simple, low-power, throughput-oriented core. Both Davis et al.[21] and Li et al. [35] have done extensive studies of the throughput-oriented core design space. We leverage this work to build our model throughput-oriented core. These efforts did examine various cache configurations, but they did not propose and evaluate caches implemented with different storage cell technologies. We show that moving from SRAM to DRAM opens up a new area of the design space to possible implementation.

The other major area of effort that we utilize in this work is work on storage cell alternatives for on-chip cache memories. The literature is rich with circuit alternatives to the traditional 6T SRAM cell. SRAM-based options seek to limit leakage effects with additional transistors [7, 15, 17, 39],

to enable standby states in the cell [32, 50, 54] or drowsy caches [33], which use microarchitectural techniques to put unused cells into low-power states. They improve upon the 6T's leakage properties, but still suffer from leakage effects.

The use of DRAM includes the traditional 1T1C cell [29] and a newer, low-latency 3T1D cell [41]. The latter does not suffer from destructive reads, but has a power footprint similar to 6T SRAM in applications that require low-latency due to refresh requirements. We employ the 1T1C cell in our work in a new application within the cache hierarchy, showing that throughput-oriented cores perform well with an L1 data cache implemented with 1T1C DRAM cells.

Finally, hybrid cache cells have been proposed [77] to provide both low leakage power and low latency performance at a reduced area to 6T SRAM. While they may indeed deliver on this promise, we find the latency characteristics of caches implemented with these cells to not be as well-matched to our throughput-oriented core's demands as the 1T1C DRAM cell-based caches are. The use of 25% 6T SRAM in a 4-way set-associative cache also limits the opportunity to increase capacity under a fixed power budget. If we consider the use of one of the mentioned SRAM cell designs that has lower leakage current, we mitigate this deficiency, but only approach the 1T1C DRAM power characteristics asymptotically.

4 Summary and Conclusions

This thesis has been motivated by the challenges presented to the microarchitect by deep submicron CMOS process technology. The availability of more transistors than we can effectively cool in a high-performance chip has caused the industry to focus on extracting value in new ways. In this work, we have looked at hard fault tolerance in the high-performance core and the data cache in a throughput-oriented design. In this chapter, we will briefly summarize our work in Section 4.1and then provide conclusions we draw from it in Section 4.2.

4.1 Summary of Results

In the first part of this work, we explored techniques to provide fine-grained, low-cost, hard-fault tolerance in the microprocessor core. The goal was to show that hard faults can be tolerated effectively within the core without macro-scale replication, thus allowing traditionally unprotected designs to adopt fault tolerance mechanisms without the traditional costs. The exchange we were willing to make for this low-cost protection was the loss of some performance and the introduction of additional complexity within the design. We proposed and evaluated three mechanisms in this space. The first two both work with array structures in the microprocessor core, while the third extended the scope of diagnosis to include all processing after instructions are decoded until their results are checked at the end of the pipeline.

In our presentation of self-repairing array structures (SRAS), our first implementation, using check rows within the array structure (SRAS-CR) utilized a small amount of redundancy within the array itself to provide for local diagnosis of faulty entries. Error detection and correction was left to an external mechanism. We selected DIVA [6] for this purpose given its relatively low cost and ease of integration into an existing commercial design. Our second implementation used error-detecting codes (EDC). In SRAS-EDC, we add delay to the access time of the array structures pro-

tected, but eliminate the need for an external detection and correction mechanism. In both SRAS-CR and SRAS-EDC, we provide the microarchitect with a tunable overhead for fault tolerance. The designer can choose how many spare array elements are available to maintain a fixed array size or, alternatively, can elect to reduce array size in the presence of failures in exchange for the consequential reduction in performance that will come from reduced capacity.

Our experimental evaluation of SRAS techniques shows that the primary advantage of SRAS is realized in structures with a high architectural vulnerability factor. This owes to the cost of error detection and correction in DIVA and related techniques, where we must flush the pipeline in order to correct a fault. In comparing SRAS-CR with SRAS-EDC, we observe that SRAS-CR can extend existing soft-fault tolerance mechanisms, such as DIVA to tolerate hard faults without great loss of performance, but that we can eliminate the need for an external detection and diagnosis mechanism altogether for a small performance penalty in the fault-free case with SRAS-EDC.

We extended the basic principles of fine-grained, low-cost hard-fault tolerance from SRAS into a processor-core global detection and diagnosis mechanism. In this design, we again relied upon DIVA for error detection and correction, but we were able to track utilization of replicated resources to identify those with hard faults present. We showed that deconfiguration of units with a high architectural vulnerability is favorable to continued error detection and correction by DIVA checkers. As with SRAS, the exploitation of redundancy within the core enables us to avoid costly recoveries in the faulted case while still retaining fault-free performance at or near parity with the unprotected design.

In our final piece of work, we presented an argument for the replacement of traditional low-latency SRAM-based data caches with lower-power, area-efficient DRAM-based data cache on the chip. To motivate our argument, we examined the demands of a throughput-oriented core and

workload on the on-chip data cache hierarchy. We then showed how a DRAM-based cache can meet the demands of this workload within a lower power envelope. We further showed that we can increase L1 data cache capacity at parity power to an SRAM-based counterpart. This nets a notable improvement in throughput over the SRAM cache. Application of these techniques to the last-level on-chip cache result in much greater power savings, opening the opportunity for a wider variety of alternatives to increase throughput at parity power, including addition of more cores as well as more cache capacity.

4.2 Conclusions

As a result of the insights this research has provided us, we draw the following three conclusions:

1) In coming generations, the trend to have more transistors than we can effectively power and cool in traditional designs will lead to an increased focus on architectural techniques that exploit this characteristic. Techniques that provide desirable features to a design while reducing hot spots and overall power density will become attractive as concerns over die area and transistor budget are viewed with less importance than power consumption. Our hard-fault tolerance mechanisms are examples of this exploitation. The extra hardware internal to the core makes for a larger overall core footprint, but the addition of storage elements for error counting and additional cold spare capacity represents die area and transistor budget consumed with a much smaller activation rate for the transistors added to the design. The net benefits of extended lifetime and part performance come at a much lower power cost than other space-consuming alternatives (e.g. triple-modular redundancy).

2) Throughput-oriented computing trends will lead microarchitects to focus on off-chip bandwidth as first-class constraints, with latency taking on a role of lesser importance in designs supporting hundreds of threads on a chip. As an extension to the work we have presented here, we conclude that this will lead to the on-chip memories' primary role moving from one of latency hiding to one of pin bandwidth buffering, with new techniques favoring bandwidth conservation favored over those in use today.

3) Our final conclusion is an insight gained from looking at both latency-sensitive and throughput-oriented problem domains. With more transistors to work with, but tighter power constraints, the use of a single core design for both latency-sensitive and throughput-oriented workloads will decline in popularity. Since many of the goals of these two compute paradigms are mutually exclusive, compromising a design to achieve characteristics desirable to both domains becomes ever more inefficient as we progress to smaller device geometries.

The work of the microarchitect is one of balancing demands and constraints. Technological breakthroughs challenge us to revisit past design decisions and examine them from a different perspective. As we move into deep sub-micron CMOS technology nodes, we see an increasing number of device manufacturing and process challenges for which no known solution exists. This increasing set of challenges will continue to provide the computer architecture community with a ready set of new problems to work on and old solutions to re-assess in new ways.

References

[1] M. Abramovici, M. A. Breuer, and A. D. Friedman. *Digital Systems Testing and Testable Design*. IEEE Press, 1990.

[2] N. Aggarwal, P. Ranganathan, N. Jouppi, and J. Smith. Configurable Isolation: Building High Availability Systems with Commodity Multicore Processors. In *Proc. of the 34th Annual Int'l Symposium on Computer Architecture (ISCA-34)*, pages 470–481, June 2007.

[3] N. Aggarwal, P. Ranganathan, N. Jouppi, and J. Smith. Isolation in Commodity Multicore Processors. *Computer*, 40(6):49–59, June 2007.

[4] AMD. Software Optimization Guide for AMD64 Processors. Publication 25112, Rev. 3.06, Sept. 2005.

[5] T. Austin, E. Larson, and D. Ernst. SimpleScalar: An Infrastructure for Computer System Modeling. *IEEE Computer*, 35(2):59–67, Feb. 2002.

[6] T. M. Austin. DIVA: A Reliable Substrate for Deep Submicron Microarchitecture Design. In *Proc. of the 32nd Annual IEEE/ACM Int'l Symposium on Microarchitecture*, pages 196–207, Nov. 1999.

[7] N. Azizi, A. Moshovos, and F. Najm. Low-Leakage Asymmetric-Cell SRAM. In *Proc. of the 2002 Int'l Symposium on Low Power Electronics and Design (ISLPED'02)*, pages 48–51, Aug. 2002.

[8] T. S. Barnett, A. D. Singh, and V. P. Nelson. Extending Integrated-Circuit Yield-Models to Estimate Early-Life Reliability. *IEEE Trans. on Reliability*, 52(3):296–300, Sept. 2003.

[9] L. Barroso, K. Gharachorloo, R. McNamara, A. Nowatzyk, S. Qadeer, B. Sano, S. Smith, R. Stets, and B. Verghese. Piranha: a scalable architecture based on single-chip multiprocessing. In *Proc. of 27th Annual Int'l Symposium on Computer Architecture (ISCA '00)*, pages 282–293, June 2000.

[10] J. M. Berger. A Note on Error Detecting Codes for Asymmetric Channels. *Information and Control*, 4:68–73, Mar. 1961.

[11] D. T. Blaauw, C. Oh, V. Zolotov, and A. Dasgupta. Static Electromigration Analysis for On-Chip Signal Interconnects. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 22(1):39–48, Jan. 2003.

[12] J. R. Black. Electromigration Failure Modes in Aluminum Metallization for Semiconductor Devices. *Proc. of the IEEE*, 57(9), Sept. 1969.

[13] R. Blish et al. Critical Reliability Challenges for The Int'l Technology Roadmap for Semiconductors (ITRS). Technical Report 03024377A-TR, Int'l SEMATECH, Mar. 2003.

[14] D. Boggs et al. The Microarchitecture of the Intel Pentium 4 Processor on 90nm Technology. *Intel Technology Journal*, 8(1), Feb. 2004.

[15] B. Calhoun and A. Chandrakasan. A 256kb Sub-threshold SRAM in 65nm CMOS. In *Digest of Technical Papers 2006 IEEE Int'l Solid-State Circuits Conference (ISSCC 2006)*, pages 2592–2601, Feb. 2006.

[16] J. R. Carter, S. Ozev, and D. J. Sorin. Circuit-Level Modeling for Concurrent Testing of Operational Defects due to Gate Oxide Breakdown. In *Proc. of Design, Automation, and Test in Europe (DATE)*, pages 300–305, Mar. 2005.

[17] L. Chang, D. Fried, J. Hergenrother, J. Sleight, R. Denard, R. Montoye, L. Sekaric, S. McNab, A. Topol, C. Adams, K. Guarini, and W. Haensch. Stable SRAM cell design for the 32 nm node and beyond. In *Digest of Technical Papers 2005 Symposium on VLSI Technology*, pages 128–129, June 2005.

[18] T. Chen and G. Sunada. A Self-Testing and Self-Repairing Structure for Ultra-Large Capacity Memories. In *Proc. of the Int'l Test Conference*, pages 623–631, Oct. 1992.

[19] T. Chen and G. Sunada. An Ultra-Large Capacity Single-Chip Memory Architecture with Self-Testing and Self-Repairing. In *Proc. of the Int'l Conference on Computer Design (ICCD)*, pages 576–581, Oct. 1992.

[20] W. B. Culbertson, R. Amerson, R. J. Carter, P. Kuekes, and G. Snider. The Teramac Custom Computer: Extending the Limits with Defect Tolerance. In *Proc. of the IEEE Int'l Symposium on Defect and Fault Tolerance in VLSI Systems*, Nov. 1996.

[21] J. Davis, J. Laudon, and K. Olukotun. Maximizing CMP Throughput with Mediocre Cores. In *Proc. of 14th Int'l Conference on Parallel Architectures and Compilation Techniques (PACT 2005)*, pages 51–62, Sept. 2005.

[22] T. J. Dell. A White Paper on the Benefits of Chipkill-Correct ECC for PC Server Main Memory. IBM Microelectronics Division Whitepaper, Nov. 1997.

[23] D. J. Dumin. *Oxide Reliability: A Summary of Silicon Oxide Wearout, Breakdown and Reliability*. World Scientific Publications, 2002.

[24] L. Gwennap. Alpha 21364 to Ease Memory Bottleneck. *Microprocessor Report*, Oct. 1998.

[25] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel. The Microarchitecture of the Pentium 4 Processor. *Intel Technology Journal*, Feb. 2001.

[26] J. Huynh. The AMD Athlon XP Processor with 512KB L2 Cache. AMD White Paper, Feb. 2003.

[27] IBM. Enhancing IBM Netfinity Server Reliability: IBM Chipkill Memory. IBM Whitepaper, Feb. 1999.

[28] Int'l Technology Roadmap for Semiconductors, 2003.

[29] S. Iyer, J. Barth, P. Parries, J. Norum, J. Rice, L. Logan, and D. Hoyniak. Embedded DRAM Technology Platform for the Blue Gene/L Chip. *IBM Journal of Research and Development*, 49(2/3):333–350, March/May 2005.

[30] JEDEC Solid State Technology Association. Failure Mechanisms and Models for Semiconductor Devices. JEDEC Publication JEP122-B, Aug. 2003.

[31] D. Jewett. Integrity S2: A Fault-Tolerant UNIX Platform. In *Proc. of the 21st Int'l Symposium on Fault-Tolerant Computing Systems*, pages 512–519, June 1991.

[32] C. Kim and K. Roy. Dynamic Vt SRAM: a leakage tolerant cache memory for low voltage microprocessors. In *Proc. of the 2002 Int'l Symposium on Low Power Electronics and Design (ISLPED'02)*, pages 251–254, Aug. 2002.

[33] N. Kim, K. Flautner, D. Blaauw, and T. Mudge. Circuit and Microarchitectural Techniques for Reducing Cache Leakage Power. *IEEE Trans. on Very Large Scale Integration (VLSI) Systems*, 12(2):167–184, Feb. 2004.

[34] S. Krumbein. Metallic Electromigration Phenomena. *IEEE Trans. on Components, Hybrids, and Manufacturing Technology*, 11(1):5–15, Mar. 1988.

[35] Y. Li, K. Skadron, B. Lee, and D. Brooks. Quantifying Latency and Throughput Compromises in CMP Design. Technical Report CS-2006-26, University of Virginia Department of Computer Science, 2006.

[36] X. Liang, R. Canal, G. Wei, and D. Brooks. Process Variation Tolerant 3T1D-Based Cache Architectures. In *Proc. of the 40th IEEE Int'l Symposium on Microarchitecture (MICRO-40)*, Dec. 2007.

[37] X. Liang, R. Canal, G. Wei, and D. Brooks. Process Variation Tolerant Register Files Based On Dynamic Memories. In *Workshop on Architectural Support for Gigascale Integration, held with Int'l Symposium on Computer Architecture (ISCA-34)*, June 2007.

[38] X. Liang, R. Canal, G. Wei, and D. Brooks. Replacing 6T SRAMs with 3T1D DRAMSs in the L1 Data Cache to Combat Process Variability. *IEEE MICRO*, 28(1):60–68, January/February 2008.

[39] S. Lin, Y.-B. Kim, and F. Lombardi. A 32nm SRAM Design for Low Power and High Stability. In *Proc. of the 51st Midwest Symposium on Circuits and Systems (MWSCAS 2008)*, pages 422–425, Aug. 2008.

[40] B. P. Linder, J. H. Stathis, D. J. Frank, S. Lombardo, and A. Vayshenker. Growth and Scaling of Oxide Conduction After Breakdown. In *41st Annual IEEE Int'l Reliability Physics Symposium Proc.* , pages 402–405, Mar. 2003.

[41] W. Luk, J. Cai, M. Immediato, and S. Kosonocky. A 3-Transistor DRAM Cell with Gated Diode for Enhanced Speed and Retention Time. In *2006 Technical Symposium on VLSI Circuits*, pages 184–185, June 2006.

[42] P. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, F. Larsson, A. Moestdedt, and B. Werner. Simics: A full system simulation platform. *IEEE Computer*, 35(2):50–58, Feb. 2002.

[43] M. Martin, D. Sorin, M. Beckman, M. Marty, A. Xu, A. Alameldeen, M. Moore, M. Hill, and D. Wood. Multifacet's General Execution-Driven Multiprocessor Simulator (GEMS) Toolset. *Computer Architecture News*, 33(4):92–99, Sept. 2005.

[44] P. Mazumder and J. S. Yih. A Novel Built-In Self-Repair Approach to VLSI Memory Yield Enhancement. In *Proc. of the Int'l Test Conference*, pages 833–841, 1990.

[45] S. McFarling. Combining Branch Predictors. Technical Report TN-36, Digital Western Research Laboratory, June 1993.

[46] G. Moore. Cramming More Components onto Integrated Circuits. *Electronics*, 38(8), 1965.

[47] S. S. Mukherjee, M. Kontz, and S. K. Reinhardt. Detailed Design and Implementation of Redundant Multithreading Alternatives. In *Proc. of the 29th Annual Int'l Symposium on Computer Architecture*, pages 99–110, May 2002.

[48] S. S. Mukherjee, C. Weaver, J. Emer, S. K. Reinhardt, and T. Austin. A Systematic Methodology to Compute the Architectural Vulnerability Factors for a High-Performance Microprocessor. In *Proc. of the 36th Annual IEEE/ACM Int'l Symposium on Microarchitecture*, Dec. 2003.

[49] M. Nicolaidis, N. Achouri, and S. Boutobza. Dynamic Data-bit Memory Built-In Self-Repair. In *Proc. of the Int'l Conference on Computer Aided Design*, pages 588–594, Nov. 2003.

[50] K. Nii, Y. Tsukamoto, T. Yoshizawa, S. Imaoka, Y. Yamagami, T. Susuki, A. Shibayama, H. Makino, and S. Iwade. A 90-nm low-power 32-kB embedded SRAM with gate leakage suppression circuit for mobile applications. *IEEE Journal of Solid State Circuits*, 39(4):684–693, Apr. 2004.

[51] K. Nikolic, A. Sadek, and M. Forshaw. Fault-Tolerant Techniques for Nanocomputers. *Nanotechnology*, 13:357–362, 2002.

[52] D. A. Patterson, G. Gibson, and R. H. Katz. A Case for Redundant Arrays of Inexpensive Disks (RAID). In *Proc. of 1988 ACM SIGMOD Conference*, pages 109–116, June 1988.

[53] I. Pomeranz and S. M. Reddy. On n-detection Test Sets and Variable n-detection Test Sets

for Transition Faults. In *Proc. of the 17th IEEE VLSI Test Symposium*, pages 173–180, Apr. 1999.

[54] H. Qin, Y. Cao, D. Markovic, A. Vladimirescu, and J. Rabaey. SRAM Leakage Suppression by Minimizing Standby Supply Voltage. In *Proc. of the 5th Int'l Symposium on Quality Electronic Design (ISQED'04)*, pages 55–60, Mar. 2004.

[55] S. K. Reinhardt and S. S. Mukherjee. Transient Fault Detection via Simultaneous Multithreading. In *Proc. of the 27th Annual Int'l Symposium on Computer Architecture*, pages 25–36, June 2000.

[56] W. C. Riordan, R. Miller, J. M. Sherman, and J. Hicks. Microprocessor Reliability Performance as a Function of Die Location for a 0.25um, Five Layer Metal CMOS Logic Process. In *Proc. of the 37th Annual IEEE Int'l Reliability Physics Symposium*, pages 1–11, Mar. 1999.

[57] R. Rodriguez, R. V. Joshi, J. H. Stathis, and C. T. Chuang. Oxide Breakdown Model and Its Impact on SRAM Cell Functionality. In *Simulation of Semiconductor Processes and Devices (SISPAD)*, pages 283–286, Sept. 2003.

[58] E. Rotenberg. AR-SMT: A Microarchitectural Approach to Fault Tolerance in Microprocessors. In *Proc. of the 29th Int'l Symposium on Fault-Tolerant Computing Systems*, pages 84–91, June 1999.

[59] K. Sawada, T. Sakurai, Y. Uchino, and K. Yamada. Built-in Self Repair Circuit for High Density ASMIC. In *Proc. of the IEEE Custom Integrated Circuits Conference*, 1989.

[60] J. Saxena et al. Scan-Based Transition Fault Testing - Implementation and Low Cost Test Challenges. In *Proc. of the Int'l Test Conference*, pages 1120–1129, Oct. 2002.

[61] E. Schuchman and T. N. Vijaykumar. Rescue: A Microarchitecture for Testability and Defect Tolerance. In *Proc. of the 32nd Annual Int'l Symposium on Computer Architecture*, pages 160–171, June 2005.

[62] M. Shah, J. Barren, J. Brooks, R. Golla, G. Grohoski, N. Gura, R. Hetherington, P. Jordan, M. Luttrell, C. Olson, B. Sana, D. Sheehan, L. Spracklen, and A. Wynn. UltraSPARC T2: A Highly-Threaded, Power-Efficient, SPARC SOC. In *Proc. of the IEEE Asian Solid-State Circuits Conference (ASSCC '07)*, pages 22–25, Nov. 2007.

[63] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically Characterizing Large Scale Program Behavior. In *Proc. of the Tenth Int'l Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 2002.

[64] P. Shivakumar, S. W. Keckler, C. R. Moore, and D. Burger. Exploiting Microarchitectural Redundancy For Defect Tolerance. In *Proc. of the 21st Int'l Conference on Computer Design*, Oct. 2003.

[65] J. E. Smith. A Study of Branch Prediction Strategies. In *Proc. of the 8th Annual Symposium on Computer Architecture*, pages 135–148, May 1981.

[66] L. Spainhower and T. A. Gregg. IBM S/390 Parallel Enterprise Server G5 Fault Tolerance: A Historical Perspective. *IBM Journal of Research and Development*, 43(5/6), September/November 1999.

[67] J. Srinivasan, S. V. Adve, P. Bose, and J. A. Rivers. The Case for Lifetime Reliability-Aware Microprocessors. In *Proc. of the 31st Annual Int'l Symposium on Computer Architecture*, June 2004.

[68] J. Srinivasan, S. V. Adve, P. Bose, and J. A. Rivers. The Impact of Technology Scaling on Lifetime Reliability. In *Proc. of the Int'l Conference on Dependable Systems and Networks*, June 2004.

[69] J. Srinivasan, S. V. Adve, P. Bose, and J. A. Rivers. Exploiting Structural Duplication for Lifetime Reliability Enhancement. In *Proc. of the 32nd Annual Int'l Symposium on Computer Architecture*, June 2005.

[70] J. R. Srour, D. Long, D. Millward, R. L. Fitzwilson, and W. L. Chadsey. *Radiation Effects on and Dose Enhancement of Electronic Materials*. Noyes Publications, 1984.

[71] K. Sundaramoorthy, Z. Purser, and E. Rotenberg. Slipstream Processors: Improving both Performance and Fault Tolerance. In *Proc. of the Ninth Int'l Conference on Architectural Support for Programming Languages and Operating Systems*, pages 257–268, Nov. 2000.

[72] J. Tao, J. F. Chen, N. W. Cheung, and C. Hu. Modeling and Characterization of Electromigration Failures Under Bidirectional Current Stress. *IEEE Trans. on Electron Devices*, 43(5):800–808, May 1996.

[73] S. Thompson et al. An Enhanced 130nm Generation Logic Technology Featuring 60nm Transistors for High Performance and Low Power at 0.7-1.4V. In *Proc. of the Int'l Electron Devices Meeting*, pages 257–260, Dec. 2001.

[74] S. Thoziyoor, N. Muralimanohar, J. Ahn, and N. Jouppi. CACTI 5.1. Technical Report HPL-2008-20, Hewlett-Packard Laboratories, Apr. 2008.

[75] C.-W. Tseng and E. J. McCluskey. Multiple-Output Propagation Transition Fault Test. In *Proc. of the Int'l Test Conference*, pages 358–366, Nov. 2001.

[76] D. M. Tullsen, S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, and R. L. Stamm. Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor. In *Proc. of the 23rd Annual Int'l Symposium on Computer Architecture*, pages 191–202, May 1996.

[77] A. Valero, J. Sahuquillo, S. Petit, V. Lorente, R. Canal, P. Lopez, and J. Duato. An Hybrid eDRAM/SRAM Macrocell to Implement First-Level Data Caches. In *Proc. of the 42nd IEEE Int'l Conference on Microarchitecture (MICRO 42)*, pages 213–221, Dec. 2009.

[78] T. N. Vijaykumar, I. Pomeranz, and K. K. Chung. Transient Fault Recovery Using Simultaneous Multithreading. In *Proc. of the 29th Annual Int'l Symposium on Computer Architecture*, pages 87–98, May 2002.

[79] J. F. Wakerly. *Error Detecting Codes, Self-Checking Circuits and Applications*. North-Holland, 1978.

[80] C. Weaver and T. Austin. A Fault Tolerant Approach to Microprocessor Design. In *Proc. of the Int'l Conference on Dependable Systems and Networks*, pages 411–420, July 2001.

[81] D. Weiss, J. J. Wuu, and V. Chin. The On-Chip 3MB Subarray Based 3rd Level Cache on an Itanium Microprocessor. In *Proc. of the Int'l Solid-State Circuits Conference*, pages 112–113, Feb. 2002.

[82] D. Wilson. The Stratus Computer System. In *Resilient Computer Systems*, pages 208–231, 1985.

[83] T.-Y. Yeh and Y. Patt. Two-level Adaptive Training Branch Prediction. In *Proc. of the 24th*

Annual IEEE/ACM Int'l Symposium on Microarchitecture, pages 51–61, Nov. 1991.

[84] L. Youngs and S. Paramanandam. Mapping and Repairing Embedded-Memory Defects. *IEEE Design & Test of Computers*, pages 18–24, January-March 1997.

Biography

Fred Allison Bower III was born September 3, 1972 in Florence, Oregon, United States of America. He received his B.S. in Mechanical Engineering and B.S. in Computer Science from Oregon State University in 1996. He received his M.S. in Computer Science and Engineering from The Oregon Graduate Institute of Science and Technology in 1999. He has published the following in pursuit of his research for this dissertation:

- Tolerating Hard Faults in Microprocessor Array Structures
- Autonomic Microprocessor Execution via Self-Repairing Arrays
- A Mechanism for Online Diagnosis of Hard Faults in Microprocessors
- Applying Architectural Vulnerability Analysis to Hard Faults in the Microprocessor
- Online Diagnosis of Hard Faults in Microprocessors
- The Impact of Dynamically Heterogeneous Multicore Processors on Thread Scheduling

Fred is a full-time employee in IBM's Systems and Technology Group where he is a Senior Technical Staff Member in System x Software Architecture.