Delivering Affordable Fault-tolerance to Commodity Computer Systems

by

Shuguang Feng

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
2011

Doctoral Committee:

Associate Professor Scott Mahlke, Chair
Professor David Blaauw
Professor Trevor N. Mudge
Professor Dennis M. Sylvester
Assistant Professor Thomas Wenisch
Pradip Bose, IBM T.J. Watson

To my parents Jinan and Pinfang.

ACKNOWLEDGEMENTS

Over the last few years there have been many moments when I felt that this dissertation would never be written. The fact that I can now place the finishing touches on this document means that there is no shortage of people to whom I owe a great debt of gratitude. Since I cannot possibly list all those that have blessed my time in Ann Arbor, I encourage everyone that I have omitted to track me down, so that I might apologize over a meal the next time we meet.

First I would like to thank my advisor, Scott Mahlke. His expert guidance and passion for his work were crucial to my success. The existence of this dissertation owes itself, in large part, to Scott's drive and unshakable optimism, which on many occasions resurrected research directions I had already resigned to abandon. Although we often disagreed, his eagerness to debate ideas and willingness to entertain dissent left an indelible mark on my approach to problem solving.

I also want to thank Jason Blome and Eric Karl who played an instrumental role in the early months of my graduate career. Their advice and guidance as I took my initial steps into academic research was invaluable.

Next, I would like to thank Shantanu Gupta and Amin Ansari. Having endured countless hours of meetings, and endless rounds of paper revisions, these two have become more

than simply co-authors. Collaborating on nearly all of our work, we have witnessed each others' failures as well as shared in each others' successes.

I would also like to thank the members (and honorary members) of the CCCP research group. I have often felt that graduate school was as much about maintaining one's sanity as it was about conducting innovative research. Whether it was our infamous coffee breaks, or seemingly random office discussions ranging from Middle East politics to Russell Peters, my labmates provided plenty of much-needed distractions.

I am also grateful to the amazing friends I made through Harvest and Knox. It is hard for me to imagine being able to survive the last few years without these friendships to remind me that there does actually exist a world outside of the office. I would especially like to thank Greg ("Magnum Opus") Davidson, Erik Kim, Manoj Cheriyan, and Matthew Remy for sharing their Ph.D. experiences with me, and showing me that, despite what trials may come, it is always possible to remember who holds tomorrow.

I also want to thank Grace Hwang for putting up with me for the last four years. Despite not seeing each other for weeks, sometimes months, at a time, she has been an encouragement throughout this Ph.D. journey. Her patience with me during particularly stressful weeks was a wonderful comfort, allowing me to selfishly vent over the phone while keeping things in perspective. Although never ideal, Grace made juggling a long-distance relationship as painless as I had any right to hope for.

Finally, I would like to thank my parents. They have sacrificed so much for me and my sister. I may never truly appreciate, much less acknowledge, the extent of their love and commitment to us. The chance to write this thesis is simply one in a long series of opportunities afforded to me because of their sacrifice.

TABLE OF CONTENTS

LIST OF FIGURES

Figure

LIST OF TABLES

Table

ABSTRACT

Delivering Affordable Fault-tolerance to Commodity Computer Systems

by

Shuguang Feng

Chair: Scott Mahlke

To meet an insatiable consumer demand for greater performance at less power, silicon technology has scaled to unprecedented dimensions. This aggressive scaling has provided designers with an ever increasing budget of cheaper and faster transistors. Unfortunately, this trend has also been accompanied by a decline in individual device reliability as transistors have become increasingly susceptible to a host of threats.

With each new technology generation the challenges associated with process variation, wearout, and transient faults gain greater prominence. We are quickly approaching a new era where fault-tolerance is becoming a first-order design constraint, no longer a luxury reserved exclusively for high-reliability, mission-critical domains. Even commodity microprocessors used in mainstream computing will require protection.

However, just as the reliability needs of NASA and Apple differ dramatically, so does their ability to absorb the costs necessary to ensure fault-tolerance. Viable solutions tar-

geting commodity systems must not only recognize this fact, but must embrace it. Simply stripping down techniques developed for enterprise servers may not result in the most appropriate designs for your laptop or cellphone. The best solutions will exploit the relaxed reliability constraints of commodity systems, judiciously sacrificing a small degree of fault-tolerance to achieve far greater reductions in overhead costs.

This thesis proposes a collection of works that can be selectively mixed and matched to assemble reliability solutions tailor-fit for the commodity systems community. Although the works presented address a variety of different issues from wearout to transient faults and prevention to detection, they were all motivated by the same observation–that much of the overhead costs associated with conventional fault tolerance mechanisms are spent in pursuit of the last few "nines" of reliability. This conclusion gave rise to the philosophy permeating the chapters of this work, that summarily dismissing techniques that cannot supply *mission-critical* fault tolerance is no longer acceptable. In presenting concrete solutions to a few of the more interesting challenges—proactive wear-leveling orchestrated through intelligent job scheduling and software-only transient fault detection and recovery that exploits intrinsic computational patterns within applications—we establish fundamental principles that can be applied more broadly to formulate a comprehensive reliability strategy.

CHAPTER I

Introduction

Given the recent news coverage of the high-profile Toyota recalls and similar articles chronicling Apple's antenna woes on their newly released iPhone, the reliability, or perhaps more appropriately the *unreliability*, of computer systems has taken center stage. Although culpability in these headlining stories may not rest solely on the shoulders of faulty microprocessors, the public response to these events has highlighted the frustration that can arise when computers, and the systems they are associated with, do not function as advertised.

1.1 Dependable Computing for the Masses

With hundreds, sometimes thousands, of dollars being spent on the latest piece of consumer electronics the computers that power them are expected to perform tasks not only quickly, but also reliably. Whether they are trading stocks from a laptop or watching the latest YouTube video on an iPhone, users expect their experience to be fault-free. Although the occasional dropped call or "blue screen of death" may be overlooked, the average con-

sumer has grown accustomed to the (nearly) fault-free enjoyment of their electronic devices.

Unfortunately, the course of aggressive technology scaling being undertaken by industry is exposing new sources of unreliability and exacerbating old ones. Whether we are talking about manufacturing defects resulting in chips that are dead-on-arrival, process variation leading to dynamic heterogeneity, wearout constraining device lifetimes, or soft-errors periodically corrupting computation, the reliability threats faced by modern microprocessors are as diverse as they are challenging.

Microarchitects who were once able to defer reliability concerns to lower level circuit and process engineers are now responsible for their share of the heavy lifting. With wearout and transient faults knocking at the door of even commodity processors, microarchitects must devise new methods to ensure consumer-visible failure rates still remain imperceptibly small, without noticeably degrading performance or cutting into shrinking profit margins.

Traditionally, reliability research has focused largely on the high-performance server market. The historical gold standards in this space have been the IBM S/360 (now Z-series servers) [95] and the HP NonStop systems [14], which rely on large scale modular redundancy to provide fault tolerance. Other researchers have focused on providing fault protection using redundant multithreading [80, 74, 59, 38, 90] or specialized hardware checkers [114, 19, 55].

These simple yet elegant techniques, having served those in the mission-critical server arena for decades, are not typically practical outside this niche domain. The overheads associated with these conventional solutions are prohibitively expensive for budget-conscious

systems with less demanding reliability requirements. Although reliability cannot be completely ignored in lower-end systems, they are not usually designed to provide the "five-nines" of fault tolerance capable of sending someone safely to the moon.

In the commodity computing space, area and power are primary considerations. Consumers are not willing to pay the additional costs (in terms of hardware price, performance loss, or reduced battery lifetime) for conventional fault-tolerance schemes. At the same time, they do not demand "five-nines" of reliability, tolerating the occasional dropped phone calls, glitches in video playback, and crashes of their desktop/laptop computers (commonly caused by software bugs). The key challenge facing the consumer electronics market in future deep submicron technologies is providing just enough fault-tolerance to ensure that the effective fault rate remains at the level to which people have become accustomed. Examining how this minimal, yet *sufficient*, coverage can be achieved "on the cheap" is the goal of this thesis.

1.2 Reliability Taxonomy

The purpose of this section is to set the stage for the remainder of the thesis by providing some preliminary background. Additional, supplemental information is supplied within subsequent chapters as needed. Given the vast amount of reliability works in the literature, this section hopes to minimize potential confusion by introducing the manner in which some fundamental terms and concepts will be used within the context of this document. Although other, presumably more formal texts, may present alternatives to these defini-

tions, our intent was to embrace the most popular definitions when possible for the sake of readability, drawing distinctions and specializing terms only when absolutely necessary.

1.2.1 Threats to Reliable Computing

For the purposes of this thesis, faults are separated into two broad categories, *transient* and *permanent*. Membership in one of these two categories, although not always a hard-and-fast rule, is generally decided based on the frequency and duration of the fault event. A transient fault is typically a "rare" event that causes an error that generally does not persist, whereas a permanent fault, once it has evolved, is almost assured to manifest as frequent consistent errors. Within permanent faults this thesis is particularly interested in those faults that are caused by device *wearout*–the process by which transistors, which are fully functional at manufacture time, degrade and eventually fail over a lifetime.

Transient Faults: Probably the better understood of the two categories, transient faults, also known as soft-errors, can be caused by a variety of phenomena. Historically, the two major sources have been neutrons from cosmic radiation and alpha particles released from packaging impurities. Whatever their origin, these high-energy particles deposit additional charge when they strike that can cause a transistor to erroneously switch. In addition to these environmental culprits, transient faults can also result from an array of other sources including crosstalk and voltage and current fluctuations. Furthermore, recent proposals for high-performance, low-power designs that employ aggressive frequency scaling and even timing speculation are also emerging as prominent causes of transient faults.

Permanent Faults: Traditionally viewed as a process and manufacturing concern, permanent faults (also known as "hard faults") have caught the attention of microarchitects in

recent years. In addition to the expected manufacturing defects, designers are now faced with the prospect of in-field wearout. This most recent threat has captured significant interest because it is closely coupled with the aggressive technology scaling trend being pursued by industry. As CMOS feature sizes scale to smaller dimensions, the inability of operating voltage to scale accordingly results in dramatic increases in power and current density. Consequently, areas of high power density increase local temperatures leading to hot spots on the chip. Since most wearout mechanisms, such as gate oxide breakdown and negative bias temperature instability are all highly dependent on temperature, the occurrence of wearout-induced failures will become increasingly common in future technology generations.

1.2.2 Anatomy of Fault-tolerant Computing

Figure 1.1 is an abstract illustration of the major components of a comprehensive reliability strategy. Each of these aspects of reliability must be addressed, at least to some degree, in order to ensure fault-tolerant operation. For example, simply detecting a fault is insufficient if no mechanism is in place to recover from it. However, not all components in Figure 1.1 require complex, sophisticated solutions. In some situations, like when a transient fault is detected, an adequate recovery response may simply be to discard the affected instructions.

The relative position of each component with respect to the transient-permanent fault boundary represents the degree to which recent state-of-the-art research publications aimed at transient (permanent) faults has focused on that particular aspect of reliability. For example, papers proposing diagnosis and system repair (reconfiguration) solutions have gen-

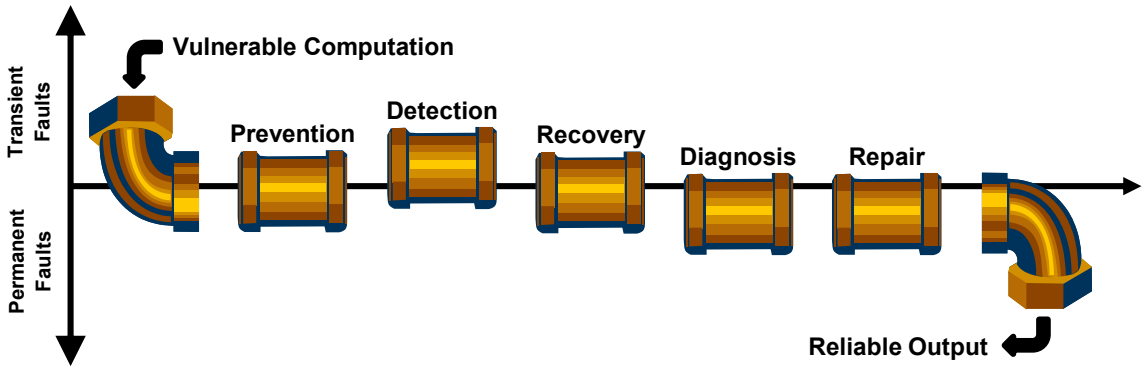


Figure 1.1: *A "reliability pipeline" depicting the different pieces of a comprehensive reliability strategy. The relative location of each component to the transient-permanent boundary represents the extent to which recent research into transient (wearout-induced permanent) faults has studied that particular aspect of reliability.*

erally been targeted toward permanent faults, mainly because transient faults typically do not damage the underlying hardware, leaving nothing to be repaired.

This thesis concentrates primarily on the first three stages in this "reliability pipeline." The first half presents techniques to monitor (detection) and proactively manage (prevention) the effects of device wearout. The later half of the dissertation tackles the challenges of transient fault detection and recovery, capitalizing on detailed program analysis to reign in overhead costs.

1.3 Conventional Solutions

As previously alluded to, processor reliability is by no means a new area of research. In as early as 1956 von Neumann formally presented the principles behind modular redundancy [63], principles that have evolved into the fault-tolerance solution of choice for nearly all application domains that have historically demanded high-reliably. In its simplest form, n-modular redundancy (nMR) relies on performing multiple instances of the same compu-

tation and employing an arbitration mechanism (typically majority voting) that can identify the correct, error-free, result from n potentially different outcomes. The archetypal fault-tolerant systems of the past have all embraced some variant of this elegant, yet effective technique [95, 11, 14].

Possibly the biggest attraction of nMR solutions is that they can nearly address all aspects of reliability (see Figure 1.1). By design they fulfill both the detection and recovery requirements, and in a simple triple-modular implementation fault diagnosis is also naturally provided (assuming a single fault model). Furthermore, given the inherent redundancy in an nMR design, even temporary repair can be easily achieved by simply ignoring the outputs of components that have been identified as faulty [1].

Yet despite the strengths of these early systems, the advent of multithreaded and multicore architectures motivated researchers to develop ways of accomplishing nMR without having to explicitly design redundancy into the hardware. Rotenberg's seminal paper on AR-SMT [80] was the first comprehensive microarchitectural-level paper on modular redundancy, introducing the concept of redundant multithreading on simultaneous multithreaded cores. Whether redundantly executing on separate cores within a CMP (spatial redundancy) or managing redundant threads running within multiple software contexts on the same hardware (temporal redundancy), later proposals like those by Vijaykumar [38, 37, 107] and Falsafi [70, 90, 91] attempted to improve upon the performance of AR-SMT by exploiting the redundant, and often idle, resources of modern superscalar CMPs.

[1] Clearly, depending on the needs of the system the faulty components would ultimately need to be replaced or the reductions in redundancy would eventually degrade reliability.

Relative to these coarse-grained redundancy approaches, the techniques proposed in this thesis incur considerably less overheads. They are not dependent on the abundance of idle processors or hardware resources. In scenarios where systems are not significantly overprovisioned, the overheads of traditional nMR are considerably greater than the few percentage points of performance degradation regularly reported in research papers. The additional resources required to enable redundant execution (i.e., additional cores and threads) must also be accounted for, resources that can no longer be allocated to other waiting tasks, adversely impacting their performance.

In contrast, the solution we propose for transient fault detection and recovery only imposes a modest performance degradation, limited to just the application being protected, without any modifications to commodity hardware. Similarly, although our wearout proposals do involve microarchitectural additions, they do not require that entire cores be reappropriated for continuous fault monitoring. Furthermore, unlike nMR solutions that can only react to failures, the wearout management techniques in this thesis are preventative and can actually proactively avoid failures.

Of course, the body of reliability literature is not solely devoted to nMR. The considerable research effort devoted toward further reducing the overheads of fault-tolerant computing has resulted in many innovative solutions that span the spectrum of the software-hardware stack. They range from compiler-directed instruction duplication [75] and specialized hardware checkers [55] for transient fault detection to adaptive body biasing and voltage scaling [105, 85] to keep wearout at bay. However, as less conventional solutions that typically only target a single component of the "reliability pipeline" (Figure 1.1) they will be discussed separately within the appropriate chapters. A more extensive treatment

of the latest advances in fault-tolerant architecture can also be found in Sorin's recent synthesis lecture [93].

1.4 Contributions

This thesis is built upon two central principles: 1) that the majority of consumer devices do not lie at either extreme of the reliability spectrum, necessitating the need for not just "low-cost" but genuinely affordable fault-tolerance, and 2) that with the appropriate analysis, the inherent computational patterns within programs can be leveraged to reap dramatic reductions in the cost of dependable computing.

With these themes in mind we make the following contributions:

- We demonstrate that the progressive nature of prominent silicon wearout mechanisms makes them amenable to low-cost, in-situ monitoring, proposing a microarchitectural-level sensor capable of tracking the evolution of wearout.
- We present a proactive reliability-aware scheduler that leverages continuous health monitoring to orchestrate application-driven wear-leveling to maximize lifetime reliability.
- We develop a commodity-grade (adequate coverage at ultra-low cost) transient fault detection mechanism that relies on reliability-aware compiler analyses to direct selective instruction duplication of vulnerable computations.
- Lastly, we present a software-only, fine-grained rollback recovery mechanism targeted at low-end commodity processors lacking native hardware recovery support.

CHAPTER II

Self-calibrating Online Wearout Detection

2.1 Introduction

Traditionally, microprocessors have been designed with worst case operating conditions in mind. To this end, manufacturers have employed burn in, guard bands, and speed binning to ensure that processors will meet a predefined lifetime qualification, or mean time to failure (MTTF). However, projections of current technology trends indicate that these techniques are unlikely to satisfy reliability requirements in future technology generations [18]. As CMOS feature sizes scale to smaller dimensions, the inability of operating voltage to scale accordingly results in dramatic increases in power and current density. Areas of high power density increase local temperatures leading to hot spots on the chip [88]. Since most wearout mechanisms, such as gate oxide breakdown (OBD), negative bias temperature instability (NBTI), electromigration (EM), and hot carrier injection (HCI), are highly dependent on temperature, power, and current density, the occurrence of wearout-induced failures will become increasingly common in future technology generations [41].

Though the reliability of individual devices is projected to decrease, consumer expectations regarding system reliability are only likely to increase. For example, some business customers have reported downtime costs of more than $1 million per hour [4]. Further, a recent poll conducted by Gartner Research demonstrated that more than 84% of organizations rely on systems that are over five years old, and more than 50% use machines that are over ten years old [1]. Given the requisite long life expectancies of systems in the field and the high costs of in-field replacement, any technique for mitigating the amount of downtime experienced due to failed systems will prove invaluable to businesses.

In order to maintain availability in the presence of potentially unreliable components, architects and circuit designers have historically employed either error detection or failure prediction mechanisms. Error detection is used to identify failed or failing components by locating (potentially transient) pieces of incorrect state within the system. Once an error is detected, the problem is diagnosed and corrective actions may be taken. The second approach, failure prediction, supplies the system with a failure forecast allowing it to take preventative measures to avoid, or at least minimize, the impact of expected device failures.

Historically, high-end server systems have relied on error detection to provide a high degree of system reliability. Error detection is typically implemented through coarse grain replication. This replication can be conducted either in space through the use of replicated hardware [95, 14], or in time by way of redundant computation [80, 74, 70, 107, 90, 37, 75, 71]. The use of redundant hardware is costly in terms of both power and area and does not significantly increase the lifetime of the processor without additional cold-spare devices. Detection through redundancy in time is potentially less expensive but is generally limited to transient error detection unless redundant hardware is readily available.

Failure prediction techniques are typically less costly to implement, but they also face a number of challenges. One traditional approach to failure prediction is the use of canary circuits [56], designed to fail in advance of the circuits they are charged with protecting, providing an early indication that important processor structures are nearing their end of life. Canary circuits are an efficient and generic means to predict failure. However, there are a number of sensitive issues that must be addressed to deploy them effectively. For instance, the placement of these circuits is extremely important for accurate prediction, because the canary must be subjected to the same operating conditions as the circuit it is designed to monitor.

Another technique for failure prediction is the use of timing sensors that detect when circuit latency is increasing over time or has surpassed some predefined threshold [31, 15, 2]. The work presented here extends upon [15] which presented the idea of failure prediction using timing analysis and identifying degrading performance as a symptom of wearout in semiconductor devices.

Recent work by Srinivasan [98] proposes a predictive technique that monitors the dynamic activity and temperature of structures within a microprocessor in order to calculate their predicted time to failure based on analytical models. This system can then be used to swap in cold-spare structures based on these predictions. This work pioneered the idea of dynamically trading performance for reliability in order to meet a predefined lifetime qualification. Although this technique may be used to identify structures that are likely to fail in the near future, it relies on accurate analytical device wearout models and a narrow probability density function for effective predictions.

Research into the physical effects of wearout on circuits has shown that many wearout mechanisms for silicon devices are progressive over time. Further, many of these wearout mechanisms, such as EM, OBD, HCI, and NBTI, have been shown to have a negative impact on device performance [7, 51, 119, 23]. For example, a device subject to hot carrier injection (HCI) will experience drive current degradation, which leads to a decrease in switching frequency [7]. The recognition of progressive performance degradation as a precursor to wearout-induced failures creates a unique opportunity for predictive measures, which can forecast failures by dynamically analyzing the timing of logic in situ.

The work presented here proposes an online technique that detects the performance degradation caused by wearout over time in order to anticipate failures. Rather than aggressively deploying duplicate fault-checking structures or relying on analytical wearout models, an early warning system is presented that identifies the performance degradation symptomatic of wearout. As a case study, and to derive an accurate performance degradation model for subsequent simulations, detailed HSPICE simulations were performed to determine the impact of one particular wearout mechanism, OBD, on logic gates within a microprocessor core. Research of other progressive wearout mechanisms such as HCI and EM, indicates that similar effects are likely to be observed as a result of these phenomenon.

The results of this analysis are used to motivate the design of an online latency sampling unit, dubbed the wearout detection unit (WDU). The WDU is capable of measuring the signal propagation latencies for signals within microprocessor logic. This information is then sampled and filtered by a statistical analysis mechanism that accounts for anomalies in the sample stream (caused by phenomenon such as clock jitter, and power and temperature fluctuations). In this way, the WDU is able to identify significant changes in the

latency profile for a given structure and predict a device failure. Online statistical analysis allows the WDU to be self-calibrating, adapting to each structure that it monitors, making it generic enough to be reused for a variety of microarchitectural components.

Traditional studies of wearout mechanisms have focused primarily on their effects on transistor and circuit level performance, without analyzing the microarchitectural impact. To the best of our knowledge, the experiments presented in this chapter were the first such attempt in this direction. Specific contributions include:

- An HSPICE-based characterization of OBD-induced wearout
- A microarchitectural analysis of the performance impact of OBD on microprocessor logic
- A detailed simulation infrastructure for modeling the impact of wearout on an embedded processor core
- A self-calibrating WDU capable of monitoring path latencies
- A demonstration of how the WDU can be deployed to extend processor lifetime

2.2 Device-level Wearout Analysis

Though many wearout mechanisms have been shown to progressively degrade performance as transistors age [7, 119, 23], as a case study, this work focuses on the effects of one particular mechanism, gate oxide breakdown (OBD), to demonstrate how performance degradation at the device level can affect processor performance at the microarchitectural level. Due to the lack of microarchitectural models for the progressive effects of wearout, it was necessary to first model the effects at the circuit level in order to abstract them up to

the microarchitecture. The results of the modeling and abstraction are presented within this section. While this section is useful in understanding the nature of progressive wearout, readers unfamiliar with device physics may want to simply note the high-level abstraction of OBD effects presented in Figure 2.1 and move on to section 2.3.

The remainder of this section describes the transistor degradation model for OBD, based on empirical data from researchers at IBM. This section also presents an HSPICE characterization of the effects of OBD on gates in a 90 nm standard cell library from a major technology vendor.

2.2.1 Gate Oxide Breakdown

OBD, also known as time dependent dielectric breakdown (TDDB), is caused by the formation of a conductive path through the gate oxide of a CMOS transistor. The progression of OBD causes an increasing leakage current through the gate oxide of devices that eventually leads to oxide failure, rendering the device unresponsive to input stimuli [100, 57, 51]. Sune and Wu showed that there is a significant amount of time required for the OBD leakage current to reach a level capable of affecting circuit performance [100]. This suggests that there is a window of opportunity to detect the onset of OBD before oxide leakage levels compromise the operation of devices and cause timing failures.

The modeling of OBD conducted in this work is based upon the experimental results of Rodriguez et al. [78]. The change in gate oxide current resulting from OBD is modeled by the power-law expression in Equation 2.1:

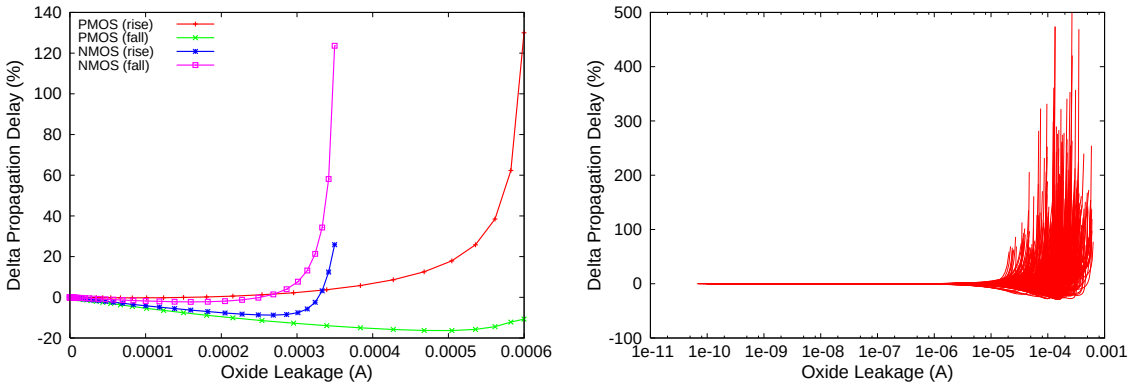
$$\Delta I_{gate} = K(V_{gd})^p \tag{2.1}$$

The change in gate-to-drain (gate-to-source) current is represented as a function of a linear scaling factor K, the gate-to-drain (gate-to-source) voltage V_{gd} (V_{gs}), and a power-law exponent, p. Both fitting parameters K and p vary depending on the amount of degradation experienced by the transistor in question. However, for much of the empirical data collected in [78], selecting a $p = 5.0$, while still allowing K to track the degree of degradation, resulted in a consistent fit. This is the model for device degradation used in this work.

2.2.2 HSPICE Analysis

To facilitate modeling the effects of OBD-induced degradation in HSPICE, the BSIM4 gate leakage model [13] for gate-to-drain and gate-to-source oxide leakage is modified to accommodate the scaling factor from Equation 2.1. Using this leakage model, an HSPICE testbench was created to simulate the effects of OBD on propagation delay within logic circuits. The testbench consists of an ideal voltage source driving an undegraded copy of the gate under test, which drives the gate under test, which drives another undegraded copy of the gate under test. This testbench allows the simulations to capture both the loading effects a degraded device presents to nodes on the upstream path, as well as the ability of downstream nodes to regenerate a degraded signal.

For each type of logic gate within the cell library, one transistor at a time is selected from the gate and its leakage model is replaced with the modified BSIM4 model. For each transistor that is being degraded, all input to output transistions are simulated so that for every gate characterized, propagation delays corresponding to all possible combinations of degraded transistor, input to output path, and initial input states are captured. For each



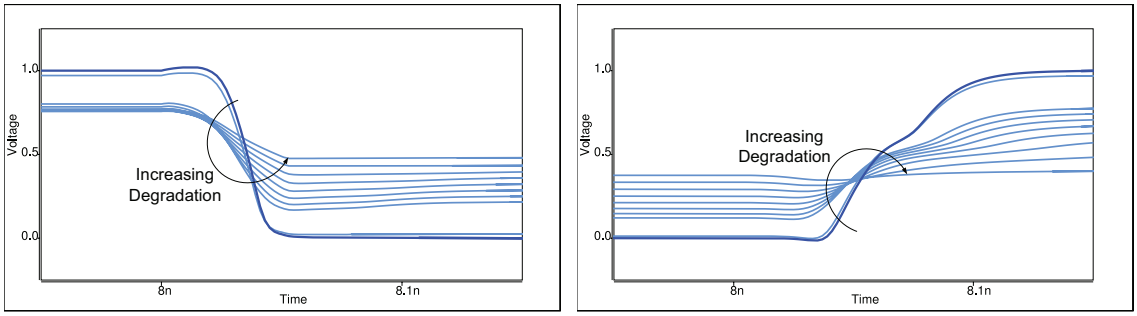
(a) Inverter delay profile.

(b) Net effect (rise+fall transitions) on propagation delay for 90 nm standard cells.

Figure 2.1: *Impact of OBD-induced oxide leakage current on standard cell propagation delays.*

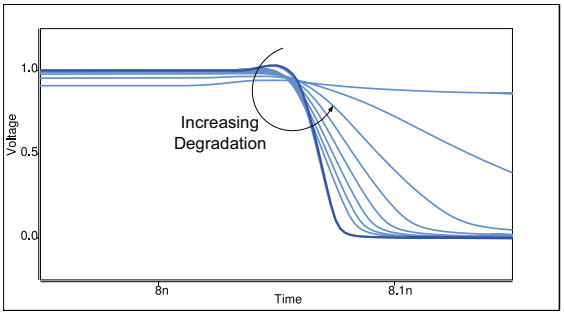
simulation run, the amount of degradation experienced by the degraded transistor (as modeled by the oxide leakage) is slowly increased until the gate ceases to function (outputs no longer switch).

The results of the timing characterization are shown in Figure 2.1. Figure 2.1a shows the changes in propagation delay for an average size inverter. The plot highlights the different effects that OBD has on propagation delay depending on the transition direction and location/type of the degraded transistor. Note that for the case when the PMOS (the analogous story is true for the NMOS) is degraded, rising transitions expressed increases in delay while falling transitions showed decreases in delay. A detailed discussion of this phenomenon follows in the next paragraph. Although there are complex dependence relationships affecting the performance impact on rise and fall propagation delays, as a simplifying assumption, the net effect is used in this work. Figure 2.1b presents the net effect (rising transition + falling transition) of OBD on gates within the cell library. For a given gate a separate curve is shown for each of its transistors. Note that the "net" change in



(a) Input to inverter cell (t_{cell}).

(b) Output of inverter cell (t_{cell}).

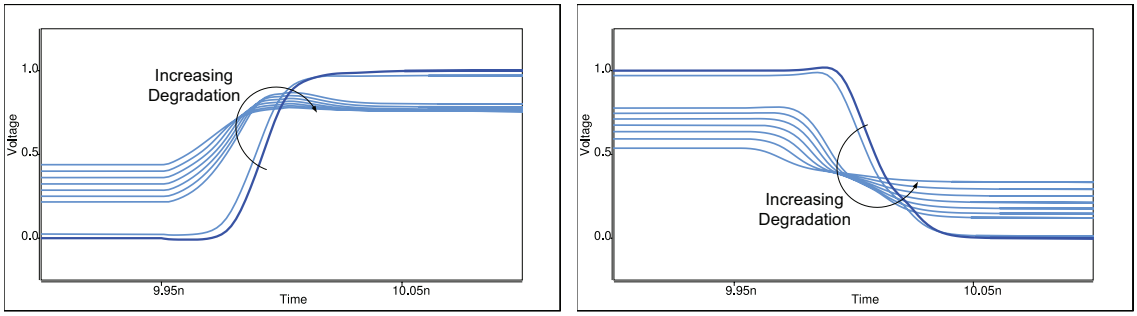


(c) Restored "circuit" output ($t_{circuit}$).

Figure 2.2: *HSPICE simulation traces for inverter with degraded PMOS (slowdown).*

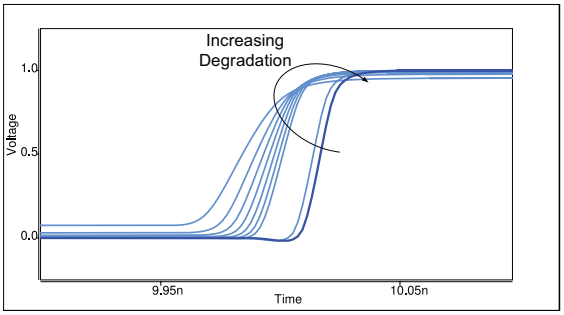
propagation delay is categorically increasing near the end of life for gates within this cell library, irrespective of which internal transistor is degraded.

An examination of Figure 2.1a reveals that in the case where the PMOS experiences OBD, the rising transition expresses more than a doubling of its nominal delay before the inverter fails to transition. The primary source of this increase in delay is the interaction with the previous stage, a non-degraded inverter, which is subjected to driving the leaky PMOS oxide. Figures 2.2 and 2.3 show the voltages at the nodes of interest during the rising and falling transitions of the degraded inverter. The bold traces show the voltage transitions under nominal conditions while the lighter curves are the result of increasing amounts of wearout.



(a) Input to inverter cell (t_{cell}).

(b) Output of inverter cell (t_{cell}).



(c) Restored "circuit" output ($t_{circuit}$).

Figure 2.3: *HSPICE simulation traces for inverter with degraded PMOS (speedup).*

When the input to the inverter under test begins to fall (Figure 2.2a), the gate-to-source oxide leakage, I_{gs}, through the PMOS device provides additional current to the input node, prolonging the discharge time of the gate through the NMOS of the preceding stage. The gate-to-drain oxide leakage, I_{gd}, initially aids the rising transition, helping to charge up the inverter output. However, as the transition continues and the output begins to rise, this I_{gd} leakage also provides an additional current to the gate node. As with the I_{gs} current, this too increases the time required to drain the remaining charge on the gate. Note also that with large amounts of degradation the input voltage range compresses due to I_{gs} and I_{gd} oxide leakage. Unable to switch from rail-to-rail, the on-currents sustainable by the PMOS and NMOS are significantly reduced, which ultimately contributes to the increase in overall propagation delay (Figure 2.2c).

Perhaps more surprising is the behavior of the propagation delay during a falling transition of the inverter output (Figure 2.3). With increasing oxide degradation, the delay of the inverter actually decreases until just prior to functional failure. This behavior is caused by the I_{gs} and I_{gd} leakage currents that help in charging their own gate node, resulting in an earlier rising transition on the input. As a result, despite the degraded on currents due to the compressed gate voltage swing, because the inverter actually "sees" the input transitioning sooner, the net effect is a decrease in the overall propagation delay of the inverter itself (t_{cell}) and ultimately the circuit ($t_{circuit}$).

In summary, at moderate values of oxide degradation, the input voltage on the gate node swings roughly rail-to-rail, allowing normal operation of the inverter. However, during the final stages of oxide OBD, the input voltage range compresses due to I_{gs} and I_{gd} leakage (Figures 2.3a and 2.2a), and the current conducted by the PMOS and NMOS devices in the inverter are significantly altered. The significantly reduced output range eventually results in functional failure when the device is no longer capable of driving subsequent stages. Note however, that prior to circuit failure, the stage immediately following the inverter under test is able to completely restore the signal to a full rail swing (Figures 2.2c and 2.3c), irrespective of the switching direction.

2.3 Microarchitecture-level Wearout Analysis

This section describes how the transistor-level models from the previous section are used to simulate the effects of OBD over time on an embedded microprocessor core. The section begins by describing the processor core studied in this work along with the synthesis

flow used in its implementation and thermal analysis. This is followed by a discussion of MTTF calculations and a description of the approach used to conduct Monte Carlo simulations of the effects of OBD. A discussion of OBD's impact on propagation delay at the microarchitectural level concludes the section.

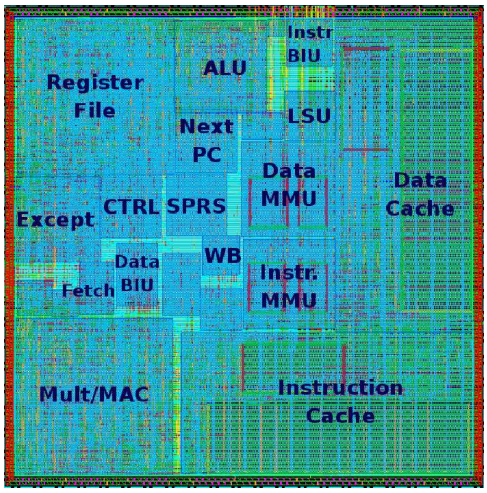
2.3.1 Microprocessor Implementation

The testbed used to conduct wearout experiments was a Verilog model of the OpenRISC 1200 (OR1200) CPU core [65]. The OR1200 is an open-source, embedded-style, 32-bit, Harvard architecture that implements the ORBIS32 instruction set. The microprocessor contains a single-issue, 5-stage pipeline, with direct mapped 8KB instruction and data caches, and virtual memory support. This microprocessor core has been used in a number of commercial products and is capable of running the μClinux operating system.

The OR1200 core was synthesized using Synopsys Design Compiler, with a cell library characterized for a 90 nm process and a clock period of 2.5 ns (400 MHz). Cadence First Encounter was used to conduct floorplanning, cell placement, clock tree synthesis, and routing. This design flow provided accurate timing information (cell and interconnect delays) and circuit parasitics (resistance and capacitance values) for the entire OR1200 core. The floorplan along with details of the implementation is shown in Figure 2.4. Note that although the OR1200 microprocessor core shown in Figure 2.4 is a relatively small design, it's area and power requirements are comparable to that of an ARM9 microprocessor. The final synthesis of the OR1200 appropriates a timing guard band of 250 ps (10% of the clock cycle time) to mimic a commodity processor and to ensure that the wearout simulations do not prematurely cause timing violations.

OR1200 Core	
Area	1.0 mm^2
Power	123.9 mW
Clock Frequency	400 MHz
Data Cache Size	8 KB
Instruction Cache Size	8 KB
Logic Cells	24,000
Technology Node	90 nm
Operating Voltage	1.0 V

(a) Implementation details for the OR1200 microprocessor.



(b) Overlay of the OR1200 floorplan on top of the placed and routed implementation of the CPU core.

Figure 2.4: *OpenRisc1200 embedded microprocessor.*

2.3.2 Power, Temperature, and MTTF Calculations

The MTTF due to OBD is dependent on many factors, the most significant being oxide thickness, operating voltage, and temperature. To quantify the MTTF of devices undergoing OBD, this work uses the empirical model described in [97], which is based on experimental data collected at IBM [117]. This model is presented in Equation 2.2.

$$MTTF_{OBD} \propto (\frac{1}{V})^{(a-bT)} e^{\frac{(X+\frac{Y}{T}+ZT)}{kT}} \tag{2.2}$$

where,

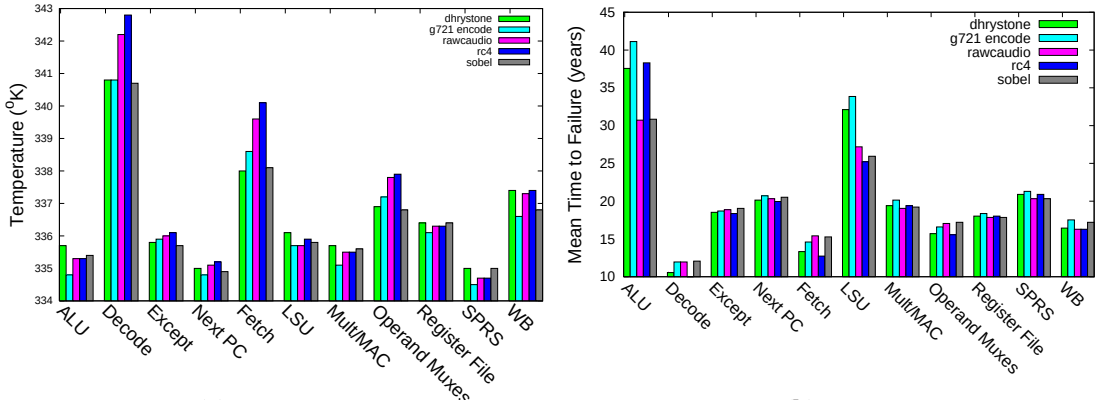
- V = operating voltage
- T = temperature
- k = Boltzmann's constant
- $a, b, X, Y,$ and Z are all fitting parameters based on [97]

In order to calculate the MTTF for devices within the OR1200 core, gate-level activity data was generated by simulating the execution of a set of benchmarks[1] running on a synthesized netlist using Synopsys VCS. This activity information, along with the parasitic data generated during placement and routing, was then used by Synopsys PrimePower to generate a per-benchmark power trace. The power trace and floorplan were in turn processed by HotSpot [88], a block level temperature analysis tool, to produce a dynamic temperature trace and a steady state temperature for each structure within the design.

Once the activity and temperature data were derived, the MTTF for each logic gate in the design was calculated using Equation 2.2 with the temperature and activity data for each benchmark. A per-module MTTF is calculated by identifying the minimum MTTF across all logic gates within each top-level module of the OR1200 core. These per-module MTTF values are later used to parametrize the statistical distribution of failures used in Monte Carlo simulations of OBD effects. Figure 2.5 presents the steady state temperatures and MTTF values of different structures within the CPU core for the five benchmarks.

Figure 2.5 highlights the correlation between MTTF and temperature. Structures with the highest temperatures tended to have the smallest MTTFs, meaning that they were most likely to wearout first. For example, the decode unit, with a maximum temperature about 3°K higher than any other structure on the chip, would likely be the first structure to fail. Somewhat surprisingly, the ALU had a relatively low temperature, resulting in a long MTTF. Upon further investigation, it was found that across most benchmark executions, less than 50% of dynamic instructions exercised the ALU, and furthermore, about 20% of

[1]Five benchmarks were studied to represent a range of computational behavior for embedded systems: dhrystone - a synthetic integer benchmark; g721encode and rawcaudio from the MediaBench suite; rc4 - an encryption algorithm; and sobel - an image edge detection algorithm.



(a) Temperature **(b)** MTTF

Figure 2.5: *Derived workload-dependent steady state temperature and MTTF for the OR1200 CPU core. An ambient temperature of 333K was used for Hotspot.*

the instructions that actually required the ALU were simple logic operations and not computationally intensive additions or subtractions. These circumstances led to a relatively low utilization and ultimately lower temperatures. It is important to note that although this work focuses on a simplified CPU model, the proposed wearout detection technique is not coupled to a particular microprocessor design or implementation, but rather relies upon the general circuit-level trends suggested by the HSPICE simulations. In fact, a more aggressive, high performance microprocessor is likely to have more dramatic hotspots, which would only serve to exaggerate the trends that motivate the WDU design presented in this work.

2.3.3 Wearout Simulation

As demonstrated in Section 2.2, progressive wearout phenomena (OBD in particular) have a significant impact on circuit-level timing. Work done by Linder and Stathis [51] has

shown that OBD-induced gate leakage obeys an exponential growth rate with age:

$$\Delta I_{OBD}(t) = I_{OBD_0} \cdot e^{t/\gamma} \tag{2.3}$$

where,

- $I_{OBD}(t)$: the amount of leakage current at time t
- I_{OBD_0}: the initial amount of leakage current at time 0
- γ: varied to model a "fast" or "slow" exponential growth

Monte Carlo simulations of the OBD effects on a distribution of microprocessors in the field are conducted by coupling the leakage model, shown in Equation 2.3, with the model for MTTF from Equation 2.2. For every logic gate within each microprocessor simulated, the time when the first initial breakdown path is formed in the oxide, τ_{BD}, is calculated using a Weibull distribution with α equal to the gate's MTTF and $\beta = 1.0$, consistent with [51]. The growth rate γ is then taken from a uniform distribution of $+/-10\%$ of τ_{BD}, consistent with a slow growth rate, as in [51].

By integrating the model for OBD failure times and this leakage growth model, a statistically accurate picture of the effects of OBD-induced leakage for every gate within the OR1200 core (across a population of chips) is derived. This new model is then used to generate age-dependent performance data for each gate within the population of processors in the Monte Carlo simulations. The performance information is then annotated onto the synthesized netlist and custom signal monitoring handlers are used to measure the signal propagation delays at the output of various modules within the design. The process of an-

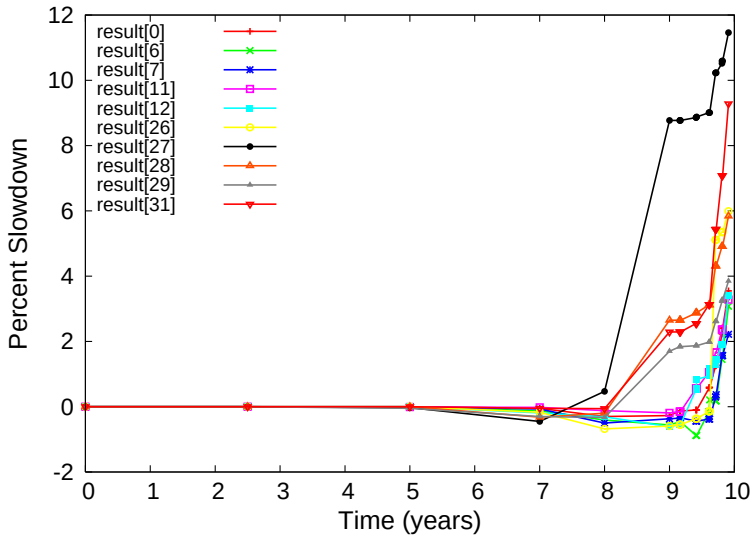


Figure 2.6: *The observed slowdown of signals from the ALU result bus as a result of OBD effects over the lifetime of one instance of an OR1200 processor core.*

notation and monitoring is repeated for every processor in the population at regular time intervals over the simulated lifetime of each processor.

To demonstrate how OBD can affect the timing of microarchitectural structures, Figure 2.6 shows the results of one sample of an OR1200 core from the Monte Carlo simulations. This figure shows the amount of performance degradation observed at the output of the ALU for a subset of signals from the result bus. This figure illustrates the general trend of slowdown across output signals from microarchitectural structures. The following section discusses how this trend is leveraged to conduct wearout detection and failure prediction.

2.4 Wearout Detection

In this section, the delay trends for microarchitectural structures observed in Section 2.3 are leveraged to propose a novel technique for predicting wearout-induced failures. The

technique consists of two logical steps: online delay monitoring and statistical analysis of delay data. In the following subsection, a circuit for conducting online delay sampling is presented. Next, the algorithm used for statistical analysis, TRIX, is presented, and its applicability to wearout detection is discussed. Finally, two potential implementations for the statistical analysis of delay profiles are proposed, one in hardware and the other in software.

2.4.1 Online Delay Profiling

In this section, a self-calibrating circuit for online delay profiling is presented. A schematic diagram of the online delay profiling unit (DPU) is shown in Figure 2.7. The DPU is used to measure the time that elapses after a circuit's output signal stabilizes until the next positive clock edge (slack time). It is important to note that even for critical paths within the design, some slack time exists because of guard bands provisioned into the design for worst-case environmental variation and signal degradation due to wearout. The DPU design consists of three distinct stages. The first stage of the DPU is an arbiter that determines which one of the (potentially many) input signals to the DPU will be profiled. The second stage of the DPU generates an approximation of the available slack time. The final stage of the DPU is an accumulator that totals a sample of 4096 signal transition latency measurements, and uses this measurement as a point estimate for the amount of available slack in the circuit for the given input signal.

The first stage fulfills the simple purpose of enabling the DPU to monitor delay information for multiple output signals from a given structure. This stage is a simple arbiter that determines which signal will be monitored. The area of this structure scales linearly

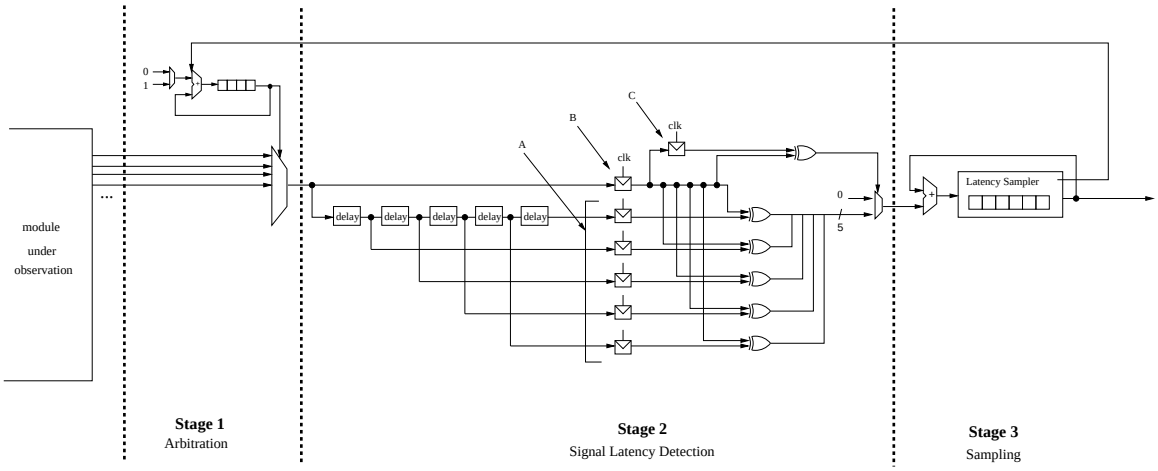


Figure 2.7: *Online delay profiling unit.*

(though very slowly) with the number of output signals being monitored. The effects of scaling on area and power are discussed later in Section 2.5.

The purpose of the second stage of the DPU is to obtain a coarse-grained profile of the amount of slack at the end of a given clock period. The signal being monitored by the DPU is connected to a series of delay buffers. Each delay buffer in this series feeds one bit in a vector of registers (labeled 'A' in Figure 2.7) such that the signal arrival time at each register in this vector is monotonically increasing. At the positive edge of the clock, some of these registers will capture the correct value of the module output, while others will store an incorrect value (the previous signal value). This situation arises because the propagation delay imposed by the sequence of delay buffers causes the output signal to arrive after the latching window for a subset of these registers. The value stored at each of the registers is then compared with a copy of the correct output value, which is stored in the register labeled 'B'. The XOR of each delayed register value with the correct value produces a bit vector that represents the propagation delay of the path exercised for that particular cycle. In addition, the output signal value from the previous cycle is stored in the register labeled

'C', and is used to identify cycles during which the module output actually experiences a transition. This ensures that cycles during which the output is idle do not bias the latency sample. As a module's performance degrades due to wearout, the signal latency seen at its outputs increases, fewer comparisons will succeed, and the value reported at the output of the vector of XOR gates will increase.

In the third stage of the DPU, a point estimate of the mean propagation latency for a given output signal is calculated by accumulating 4096 signal arrival times. The accumulation of 4096 arrival times is used to smooth out the variation in path delays that are caused by variation in the module input, and the sample size 4096 is used because it is a power of two and allows for efficient division by shifting.

There are multiple ways in which this sampled mean propagation latency may be utilized by a system for failure prediction. In the next subsection, an algorithm is presented for this purpose that may be implemented either in specialized hardware or software.

2.4.2 Failure Prediction Algorithm

In order to capitalize on the trend of divergence between the signal propagation latency observed during the early stages of the microprocessor's lifetime and those observed at the end of life, TRIX (triple-smoothed exponential moving average) [99] analysis is used. TRIX, is a trend analysis technique used to measure momentum in financial markets and relies on the composition of three calculations of an exponential moving average (EMA) [12]. The EMA is calculated by combining the current sample value with a fraction of the pre-

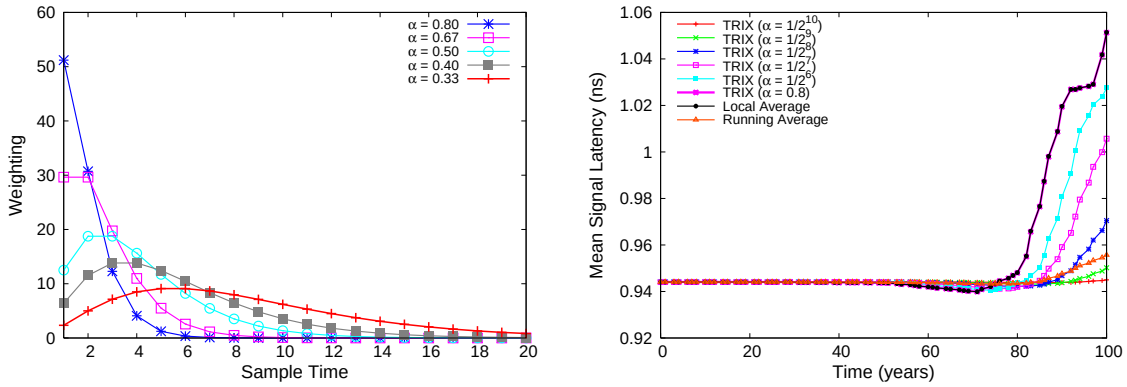
vious EMA, causing the weight of older sample values to decay exponentially over time. The calculation of EMA is given as:

$$EMA = \alpha \times sample + (1 - \alpha) EMA_{prev} \tag{2.4}$$

The use of TRIX, rather than the EMA, provides two significant benefits. First, TRIX provides an excellent filter of noise within the data stream because the composed applications of the EMA smooth out aberrant data points that may be caused by dynamic variation, such as temperature or power fluctuations (discussed in Section 2.5.2). Second, the TRIX value tends to provide a better leading indicator of sample trends. The equations for computing the TRIX value are:

$$\begin{aligned} EMA_1 &= \alpha(sample - EMA_{1_{prev}}) + EMA_{1_{prev}} \\ EMA_2 &= \alpha(EMA_1 - EMA_{2_{prev}}) + EMA_{2_{prev}} \\ TRIX &= \alpha(EMA_2 - TRIX_{prev}) + TRIX_{prev} \end{aligned} \tag{2.5}$$

TRIX calculation is recursive and parametrized by the weight, α, which dictates the amount of emphasis placed on older sample values. Figure 2.8a demonstrates the impact of different α values on the amount of weight given to historical samples. This figure demonstrates that small α values tend to favor older samples, while larger α values reflect local trends. The wearout detection algorithm presented in this work relies on the calculation of two TRIX values using different α's to identify when the local trends in the observed signal latency begin to diverge from the historical trends (biased toward early-life timing).



(a) Impact of α value on the weighting of old sample values.

(b) Impact of α value on the tracking of a signal undergoing OBD degradation effects.

Figure 2.8: *Sensitivity analysis of TRIX parametrization.*

Figure 2.8b shows the effect of different α values on the TRIX analysis of ALU result bit 0. Figure 2.8b presents the TRIX calculations for six different α values as well as the long-term running average and local point average of signal over the lifetime of the microprocessor. This data demonstrates that TRIX calculation using $\alpha = 1/2^{10}$ provides an accurate estimate of the running average (of latencies for a signal) over the lifetime of the chip, and does so without the overhead of maintaining a large history. Further, this figure shows that a TRIX calculation with $\alpha = 0.8$ provides a good indicator of the local sample latency at a given point in the microprocessor's lifetime.

The next subsection describes two potential implementations that bring together the DPU and this statistical analysis technique in order to predict the failure of structures within a processor core.

2.4.3 Implementation Details

In order to accurately detect the progression of wearout and predict when structures are likely to fail, this work proposes the use of the DPU in conjunction with TRIX analysis.

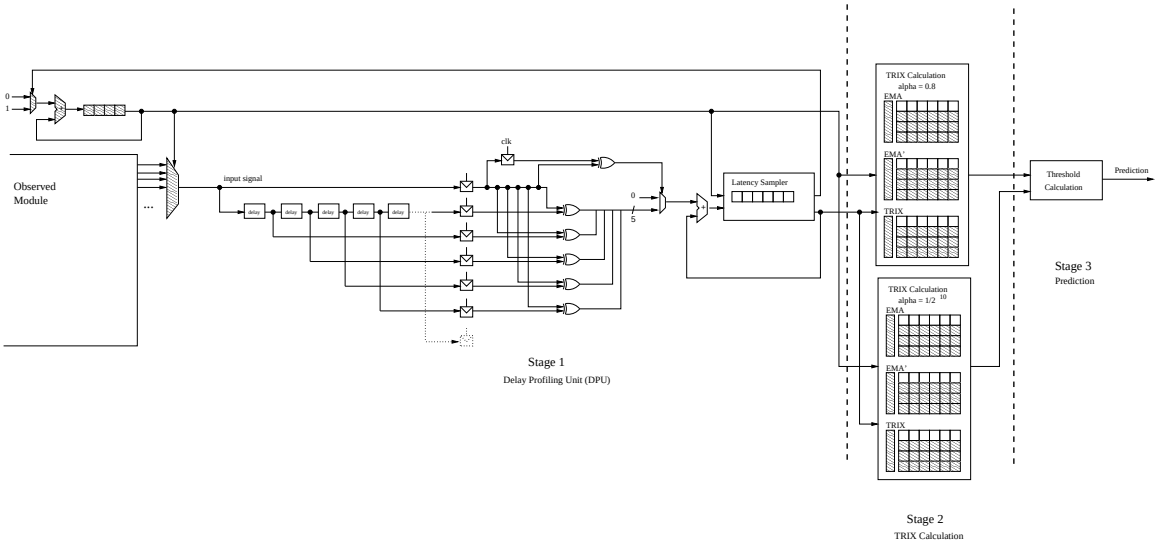


Figure 2.9: *Design and organization of the wearout detection unit.*

In the following subsections, two techniques for building systems with wearout prediction mechanisms are proposed. The first technique is a hardware-only approach, where both online delay profiling and TRIX analysis are conducted together in a specialized hardware unit called the wearout detection unit (WDU). The second technique is a hybrid approach requiring fewer resources where delay profiling is conducted in hardware, but TRIX analysis is conducted in software, either in the operating system or in firmware. In Section 2.5, we discuss the hardware costs in terms of area and power for each of these implementations, as well how the WDU scales as it is used to monitor an increasing number of signals.

2.4.3.1 Hardware-only Implementation

The design of the WDU is presented in Figure 2.9 and consists of three distinct stages. The first stage is comprised of the delay profiling unit described in Section 2.4.1, while the second stage is responsible for conducting the TRIX analysis discussed in Section 2.4.2, and the third stage conducts threshold analysis to identify significant divergences in latency

trends. The shaded structures in this diagram represent those components that would scale with the number of signals being monitored. The remainder of this section discusses the implementation details of stage two and three of this design, and the required resources for their implementation.

In stage two of the WDU, two TRIX values are computed: a locally-biased value, $TRIX_l$, and a historically-biased value, $TRIX_g$. These are calculated using α values of 0.8 and $1/2^{10}$, respectively. It is important to note that the value of α is dependent on the sample rate and sample period. In this work, we assume a sample rate of three to five samples per day over an expected 10 year lifetime. Also, the long incubation periods for many of the common wearout mechanisms require the computed TRIX values to routinely be saved into a small area of nonvolatile storage, such as flash memory.

Since the TRIX consists of three identical EMA calculations, the impact of Stage 2 on both area and power can be minimized by spanning the calculation of the TRIX values over multiple cycles and only synthesizing a single instance of the EMA calculation hardware. Section 2.5 describes the area and power overhead for the WDU in more detail.

The third stage of the WDU receives $TRIX_l$ and $TRIX_g$ values from the previous stage and is responsible for predicting a failure if the difference between these two values exceeds a given threshold. The simulations conducted in this work indicate that a 5% difference between $TRIX_l$ and $TRIX_g$ is almost universally indicative of a structure nearing failure. It is envisioned that this prediction would be used to enable a cold spare device, or notify a higher-level configuration manager of a potentially failing structure within the core. An analysis of the accuracy of this threshold prediction is presented in Section 2.5.

2.4.3.2 Hardware/Software Hybrid Implementation

In order to alleviate some of the scaling problems and resource requirements of a hardware only technique, a hardware/software hybrid technique can be used. In this system, the DPU is still implemented in hardware, while the TRIX analysis is performed in software by the operating system or system firmware. In this configuration, a set of dedicated registers for maintaining the latency samples for different modules within the design are used. These dedicated registers are similar to the performance counters used in modern day processors. The system software then regularly samples these counters and can store the calculated TRIX values to disk or other non-volatile storage.

This hardware/software hybrid design has multiple benefits over the hardware-only approach. In the hardware-only approach, the TRIX calculation, as well as the α parametrization values are hard-wired into the design, meaning that across different technology generations with different wearout progression rates, different WDU implementations will be necessary. However, in the hybrid approach, the TRIX parametrization is easily modified for use in a variety of systems. Another benefit is that the hybrid implementation consumes less power and has a smaller area footprint with better scaling properties than the hardware-only design.

2.5 Experimental Analysis

This section provides a detailed analysis of the proposed WDU for both the hardware-only and hybrid implementations, the area and power overhead for implementation, and its efficacy in predicting failure.

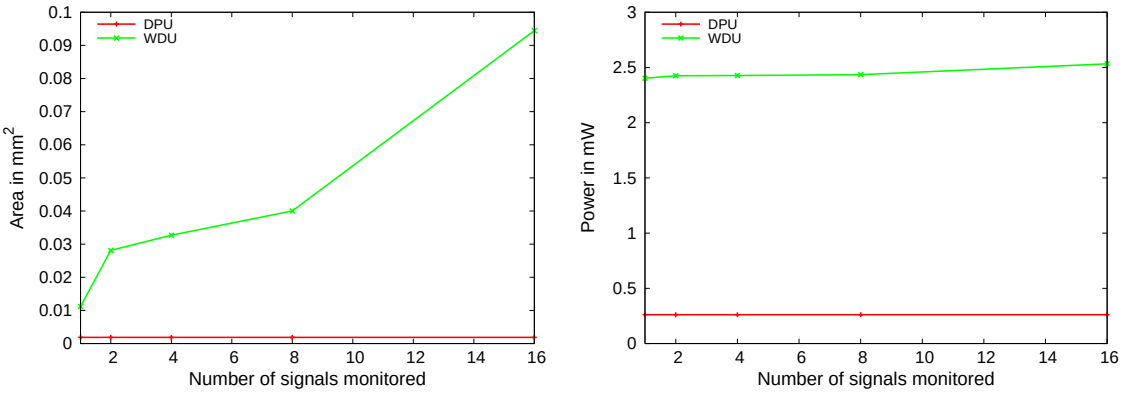
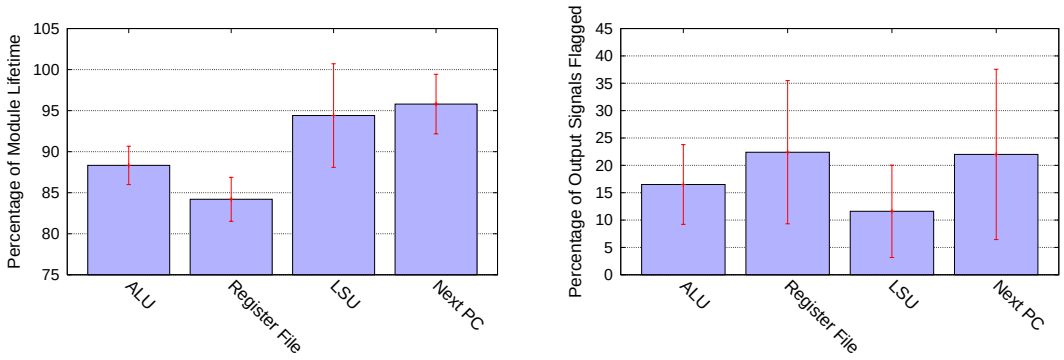


Figure 2.10: *Scaling of the WDU and DPU area and power as the number of signals monitored scales.*

2.5.1 Overhead and Accuracy

Figure 2.10 demonstrates the area and power requirements for a WDU and a DPU (for the hybrid approach) implemented in Verilog and synthesized using a 90 nm standard cell library, designed to monitor multiple output signals for a structure. The x-axis represents the number of signals being monitored and the y-axis represents the overhead in terms of area or power. Figure 2.10a demonstrates that the WDU scales poorly in terms of area, and Figure 2.10b shows analagous results for power. This behavior is largely because the amount of storage within the WDU increases linearly with the number of signals being monitored. In contrast, the DPU scales well in both area and power with an increasing number of signals being monitored because only the logic for the arbiter scales with an increasing number of signals, and this increase in logic is for the most part negligible. This implies that the hybrid prediction technique can be implemented at a much lower design cost.

In order to evaluate the efficacy of TRIX analysis in predicting failure, a large number of Monte Carlo wearout simulations were conducted using the Weibull distribution and



(a) Average percentage of the lifetime at which failure predictions are made with error bars representing the standard deviation of the population.

(b) Percentage of the output signals that were flagged at the time of failure for each module with error bars representing the standard deviation of the population.

Figure 2.11: *Analysis of TRIX analysis efficacy in predicting failure.*

failure model presented in Section 2.3.2. Figure 2.11a demonstrates the relative time at which failure was predicted for a variety of structures within the processor core for the population of microprocessors used in this Monte Carlo simulation. The error bars in this figure represent the standard deviation of these values. Across all simulations, failure was predicted within 20% of the time of failure for the device. This typically amounted to slightly less than two years of remaining life before the device ultimately failed. Two extreme cases were the Next PC module and the LSU, where the failure prediction was often almost too optimistic, with many of the failure predictions being made with only about 1% or about 4 days of the structure's life remaining. On the opposite end of the spectrum, failure of the register file was often predicted with more than 15% of the lifetime remaining, meaning that some usable life would be wasted in a cold-sparing situation.

Figure 2.11b demonstrates the percentage of signals that caused predictions to be raised for each module before the module failed. In general, the percentage of outputs flagged at the time of failure varied widely. This can be attributed to a number of factors. First, the

Weibull distribution used to model the time of first breakdown for each gate within the design has a moderate amount of variance, as does the uniform distribution used to model the growth rate of leakage from the time of first breakdown. Also, because some gates experience speedup in the early stages of wearout before they ultimately begin to slow down, there are competing effects between gates at different stages of wearout early in the breakdown period.

2.5.2 Dynamic Variations

Dynamic environmental variations, such as temperature spikes, power surges, and clock jitter, can each have an impact on circuit-level timing, potentially affecting the operation of the WDU. Here, we briefly discuss some of the sources of dynamic variation and their impact on the WDU's efficacy.

Temperature is a well known factor in calculating device delay, where higher temperatures typically increase the response time for logic cells. Figure 2.12 demonstrates the increase in response time for a selection of logic gates[2] over a wide range of temperatures. This figure shows that over an interval of 50^{o}C, the increase in response time amounts to approximately 3.4%.

Another source of variation is clock jitter. In general, there are three types of jitter: absolute jitter, period jitter, and cycle-to-cycle jitter. Of these, cycle-to-cycle jitter is the only form of jitter that may potentially affect the WDU. Cycle-to-cycle jitter is defined as the difference in length between any two adjacent clock periods and may be both positive

[2]The gate models were taken from the 90 nm library and simulated using HSPICE.

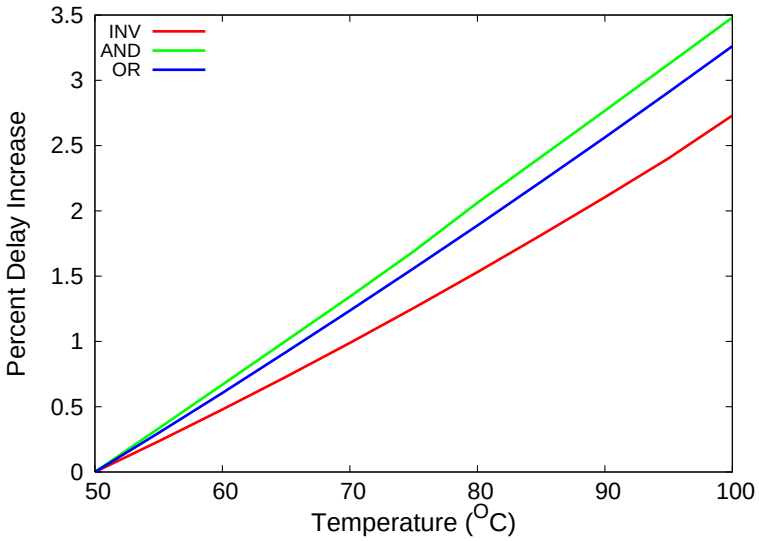


Figure 2.12: *Impact of temperature on logic gate delay.*

(cycle 2 longer than cycle 1) or negative (cycle 2 shorter than cycle 1). Statistically, jitter measurements exhibit a random distribution with a mean value approaching zero [118].

In general, the sampling techniques employed for failure prediction are sufficient to smooth out the effects of dynamic variation described. For example, a conservative, linear scaling of temperature effects on the single inverter delay to a 3.4% increase in module output delay does not present a sufficient magnitude of variance to overcome the 5% threshold required for the WDU to predict failure. Also, because the expected variation due to both clock jitter and temperature will exhibit a mean value of zero (i.e., temperature is expected to fluctuate both above and below the mean value), statistical sampling of latency values should minimize the impact of these variations. To further this point, since the TRIX calculation acts as a three-phase low-pass filter, the worst case dynamic variations would need to cause latency samples to exceed the stored $TRIX_g$ value by greater than 5% over the course of more than 12 successive sample periods, corresponding to over four days of operation.

2.6 Related Work

Issues in technology scaling and process variation have raised concerns for reliability in future microprocessor generations. Recent research work has attempted to diagnose and, in some cases, reconfigure the processing core to increase operational lifetime. In this section, we briefly discuss this related work and how it has influenced the design of the WDU.

As mentioned in Section 2.1, much of the research into failure detection relies upon redundancy, either in time or space. One such example of hardware redundancy is DIVA [8], which targets soft error detection and online correction. It strives to provide a low cost alternative to the full scale replication employed by traditional techniques like triple-modular redundancy. The system utilizes a simple in-order core to monitor the execution from a large high performance superscalar processor. The smaller checker core recomputes instructions before they commit and initiates a pipeline flush within the main processor whenever it detects an incorrect computation. Although this technique proves useful in certain contexts, the second microprocessor requires significant design/verification effort to build and incurs additional area overhead.

Bower et al. [19] extends the DIVA work by presenting a method for detecting and diagnosing hard failures using a DIVA checker. The proposed technique relies on maintaining counters for major architectural structures in the main microprocessor and associating every instance of incorrect execution detected by the DIVA checker to a particular structure. When the number of faults attributed to a particular unit exceeds a predefined threshold, it is deemed faulty and decommissioned. The system is then reconfigured, and in the presence of cold spares, can extend the useful life of the processor. Related work by Shivakumar et

al. [86] argues that even without additional spares the existing redundancy within modern processors can be exploited to tolerate defects and increase yield through reconfiguration.

Research by Vijaykumar [37, 107] and Falsafi [70, 90] attempt to exploit the redundant, and often idle, resources of a superscalar processor to enhance reliability by utilizing these extra units to verify computations during periods of low resource demand. This technique represents an example of the time redundant computation alluded to in Section 2.1. It leverages work by the Slipstream group [80, 71] on simultaneous redundant multithreading as well as earlier work on instruction reuse [92].

Srinivasan et al. have also been very active in promoting the need for robust designs that can withstand the wide variety of reliability challenges on the horizon [98]. Their work attempts to accurately model the MTTF of a device over its operating lifetime, facilitating the intelligent application of techniques like dynamic voltage and/or frequency scaling to meet reliability goals. Although some physical models are shared in common, the focus of the WDU is not to guarantee that designs can achieve any particular reliability goal, but rather to enable a design to recognize behavior that is symptomatic of wearout induced breakdown allowing it to react accordingly.

Analyzing circuit timing in order to self-tune processor clock frequencies and voltages is also a well studied area. Kehl [43] discusses a technique for re-timing circuits based on the amount of cycle-to-cycle slack existing on worst-case latency paths. The technique presented requires offline testing involving a set of stored test vectors in order to tune the clock frequency. Although the proposed circuit design is similar in nature to the WDU, it only examines the small period of time preceding a clock edge and is only concerned with worst case timing estimation, whereas the WDU employs sampling over a larger time span

in order to conduct average case timing analysis. Similarly, Razor [9] is a technique for detecting timing violations using time-delayed redundant latches to determine if operating voltages can be safely lowered. Again, this work studies only worst-case latencies for signals arriving very close to the clock edge.

2.7 Summary

In this chapter we characterized the device-level effects of oxide breakdown (OBD) on circuit performance and demonstrated that progressive OBD has a non-uniform impact on circuit performance. The results of the circuit-level modeling were then applied to a synthesized implementation of the OR-1200 microprocessor to analyze the effects of OBD at the microarchitectural level. Circuit timing was identified as a common phenomenon that can be tracked to predict the progression of OBD. A self-calibrating circuit for analyzing circuit path delay along with an algorithm for predicting failure using this data was proposed. Results show that our failure prediction algorithm is successful in identifying wearout and flagging outputs that suffer a trend of increasing delay over time.

CHAPTER III

Maestro: Orchestrating Lifetime Reliability in Chip Multiprocessors

3.1 Introduction

In recent years, computer architects have accepted the fact that transistors become less reliable with each new technology generation [18]. As technology scaling leads to higher device counts, power densities and operating temperatures will continue to rise at an alarming pace. With an exponential dependence on temperature, faults due to failure mechanisms like negative-bias temperature instability (NBTI) and time-dependent dielectric breakdown (TDDB) will result in ever-shrinking device lifetimes. Furthermore, as process variation (random + systematic) and wearout gain more prominence in future technology nodes, fundamental design assumptions will become increasingly less accurate. For example, the characteristics of a core on one part of a chip multiprocessor (CMP) may, due to manufacturing defects, only loosely resemble an *identically designed* core on a different part of the CMP [116, 103]. Even the behavior of the same core can be expected to change over time as a result of age-dependent degradation [77, 105].

In light of this uncertain landscape, researchers have begun investigating dynamic thermal and reliability management (DTM and DRM). Such techniques hope to sustain current performance improvement trends deep into the nanometer regime, while maintaining the levels of reliability and life-expectancy that consumers have come to expect, by hiding a processor's inherent susceptibility to failures and hotspots. Some recent proposals rely on a combination of thread scheduling and dynamic voltage and frequency scaling (DVFS) to recover performance lost to process variation [103, 116]. Others implement intelligent thermal management policies that can extend processor lifetimes and alleviate hotspots by minimizing and bounding the overall thermal stress experienced by a core [52, 67, 27, 24]. There have also been efforts to design sophisticated circuits that tolerate faults and adaptive pipelines with flexible timing constraints [30, 104]. Although many DTM schemes actively manipulate job-to-core assignments to avoid thermal emergencies, most existing DRM approaches only *react* to faults, tolerating them as they develop.

In contrast, Maestro takes a proactive approach to reliability. To the first order, Maestro performs fine-grained, module-level wear-leveling for many-core CMPs. Although analogous to wear-leveling in flash devices, the challenge of achieving successful wear-leveling transparently in CMPs is considerably more difficult. Left unchecked, wearout causes all structures within a core to age and eventually fail. However, due to process variation, not all cores (or structures) will be created equal. Every core will invariably possess some microarchitectural structures that are more "damaged" (more susceptible to wearout) than others [104, 103]. Performing post-mortems on failed cores (in simulations) often reveals that a single microarchitectural module, which varies from core to core, breaks down long before the rest. Maestro extends the life of these "weak" structures, their corresponding

cores, and ultimately the CMP by ensuring uniform aging with scheduling-driven wear-leveling across all levels of the hierarchy.

Maestro dynamically formulates wearout-centric schedules, where jobs are assigned to cores such that cores do not execute workloads that apply excessive stress to their weakest modules (i.e., a floating-point intensive thread is not bound to a core with a weakened floating-point adder). This accomplishes *local wear-leveling* at the core level, avoiding failures induced by a single weak structure. When two cores both have a strong affinity for the same job, a heuristic, which enforces *global wear-leveling* at the CMP level determines which core is given priority. Typically, unless there is a substantial negative impact on local wear-leveling, deference is given to the weaker of the two cores. This ensures that, when necessary, stronger cores are allowed to execute less desirable jobs in order to postpone failures in weaker cores (details in Section 3.3.2).

By leveraging the natural, module-level diversity in application thermal footprints (Section 3.2.3), Maestro has finer-grained control over the aging process than a standard core-level DVFS approach, without any of the attendant hardware/design overheads. Given the complex nature of wearout degradation, Maestro departs from the conventional reliance on static analysis to project optimized schedules. Instead, the condition of the underlying CMP hardware is continuously monitored, allowing Maestro to dynamically refine and adapt scheduling algorithms as the system ages. Architectures like those envisioned in [101], with low-level circuit sensors, can readily supply this real-time "health" monitoring.

Maestro offers two key benefits for future CMP systems. First, the fine-grained, local wear-leveling prevents unnecessary core failures, maximizing the life of *individual* cores. Longer lasting cores translates to more work that can be done over the life of the sys-

tem. Second, it improves the ability of the system to sustain heavy workloads despite the effects of aging. Enforcing global wear-leveling maximizes the *number* of functional cores (throughout its useful life), which in turn maximizes the computational horsepower available to meet peak demands. With higher degrees of process variation on the horizon, premature core failures will make it increasingly more difficult to design and qualify future CMPs. However, by harnessing the potential of Maestro, proactive management will enable semiconductor manufacturers to provide chips with longer lifetimes as well as ensure that system performance targets are consistently met throughout that lifetime. The central contributions of this chapter include:

- An evaluation of workload variability and its impact on reliability/wearout.
- An introspective system, Maestro, that utilizes low-level sensor feedback and application-driven wear-leveling to proactively manage lifetime reliability.
- The design and evaluation of two reliability-centric job scheduling algorithms.

3.2 Scheduling for Damaged Cores and Dynamic Workloads

As mentioned, researchers have investigated techniques that leverage intelligent job scheduling to recover performance, manage on-core temperatures, or cope with process variation. However, none have studied the influence that wearout-centric scheduling alone can have on the evolution of wearout within a core, and the overall lifetime reliability of a CMP system. Section 3.2.1 presents a brief overview of common failure mechanisms. Next, Section 3.2.2 surveys previously proposed scheduling approaches and highlights limitations. Then, to quantify the potential for reliability-centric scheduling, Section 3.2.3

Mechanism	Mean Time to Failure (MTTF)	Failure Mode
NBTI	$MTTF \propto (\frac{1}{V})^{\gamma} e^{\frac{E_{aNBTI}}{\kappa T}}$	Shifting V_t leads to increasing leakage current and slower devices that eventually cease to switch.
TDDB	$MTTF \propto (\frac{1}{V})^{(a-bT)} e^{\frac{(X+\frac{Y}{T}+ZT)}{\kappa T}}$	Compromises the oxide, leading to increasing gate current and switching delay.
EM	$MTTF \propto (\alpha \frac{CV}{WH})^{-n} e^{\frac{E_{aEM}}{\kappa T}}$	Accumulation/depletion of metal in interconnects results in faults due to shorts and voids.
TC	$MTTF \propto (\frac{1}{T-T_{ambient}})^{q}$	Fatigue due to thermal expansion/ contraction leads to packaging failures.

Table 3.1: *Common failure mechanisms: Negative Bias Temperature Instability (NBTI), Time Dependent Dielectric Breakdown (TDDB), Electromigration (EM) and Thermal Cycling (TC). V = voltage, T = temperature, α = switching factor, κ = Boltzmann's constant, V_t = threshold voltage, and all other variables are technology dependent fitting parameters.*

examines the module-level thermal diversity seen across a set of SPEC2000 applications. Lastly, Section 3.2.4 presents preliminary results quantifying the impact of this variation on processor lifetimes.

3.2.1 Failure Mechanism Review

A large body of work exists in the literature on characterizing the behavior of wearout mechanisms that age processors. Researchers have focused on capturing the dependence of these mechanisms on operating parameters like voltage and temperature. This dependence is typically presented in the form of mean time to failure (MTTF) equations. These equations are then often used to project the average expected lifetime of a processor in the field, given a set of worst-case operating conditions (Vdd, temperature, etc.). Designing and qualifying a processor to meet a target MTTF with worst-case (or near-worst-case) conditions in mind ($MTTF_{wc}$) ensures that expected operating lifetimes will be met

because actual operating conditions in the field tend to be much milder than worst case ($MTTF_{actual} >> MTTF_{wc}$). Table 3.1 summarizes some of the common mechanisms that have been studied in the past. Note the strong dependence of all the mechanisms on temperature.

Research into the physical effects of wearout has shown that many prominent mechanisms, especially TDDB and NBTI, are progressive in nature [51, 119, 23]. Unlike soft-errors that can occur suddenly and without warning, wearout-related faults are typically more gradual, manifesting as small defects that eventually evolve into hard faults. This property of wearout suggests that before age-induced degradation can cause permanent failures in a CMP, measuring the accumulation of damage can actually be used to dynamically monitor the life-expectancy of individual cores. The remainder of this chapter targets TDDB and NBTI, which are expected to be the two leading causes of wearout-related failures in future technologies, but can be easily extended to address any progressive failure mechanisms that may emerge in the future.

3.2.2 Existing Scheduling Schemes

Scheduling, in the context of this chapter, refers to the process of assigning jobs to cores in a CMP, and is conceptually decoupled from the operating system (OS) scheduler. The schedulers proposed by microarchitects in the past typically resided in a virtualization layer (i.e., system firmware) that sits between the OS and the underlying hardware. At each scheduling interval the OS supplies a set of jobs, J, to this virtualization layer, and it is the task of the low-level scheduler to bind the jobs to cores.

Prior work in this area can be roughly divided into two broad categories: performance-centric schedulers, *perf-centric*, which exploit the variability in performance characteristics between cores in a CMP [103, 116], and thermal-aware schedulers that are cognizant of on-chip temperatures. Work within the thermal-aware category can be further differentiated into those that target power reduction or performance improvement (higher allowable frequencies) [67, 88, 20, 46], *thermal-p*, and those that target reliability enhancement [52, 105], *thermal-r*. The characteristics, and limitations, of schedulers within each of these categories is discussed below.

3.2.2.1 perf-centric

In large CMP systems, whether due to process variation or aging (or both), some cores will have slower critical paths or even non-functional components. In this environment where some cores have better performance characteristics than others (e.g., faster sustainable frequencies or more usable issue queue entries) performance-oriented schedulers seek to identify intelligent schedules that match application requirements to hardware capability. Appropriate scheduling results in better overall system performance, and can hide to an extent, the existence of cores with degraded functionality. However, perf-centric schedulers fail to address a major root-cause of performance degradation, device aging.

3.2.2.2 thermal-p

These schedulers attempt to identify schedules that minimize peak temperatures across a core and/or CMP by exploiting the spatial locality of heat dissipation. Interleaving hot and cool jobs between adjacent cores alleviates the stress on the thermal packaging, increasing

its efficiency, by not concentrating the heat among neighboring cores. This reduces power consumption and the probability of temperatures exceeding thermal thresholds, minimizing the performance penalty incurred by thermal throttling. Oftentimes these techniques can also be coupled with DVFS. By running at higher frequencies to exploit the thermal headroom afforded by scheduler-driven hotspot mitigation, and relying on judicious DVFS to avoid thermal emergencies, overall system performance can be greatly enhanced. Although minimizing peak temperatures improves the reliability of the hottest processor structures, many core failures are caused by modules that only experience moderate temperatures but were more prone to aging due to process variation.

3.2.2.3 thermal-r

In a slight variation on thermal-p schedulers, these techniques attempt to track the effects of thermal stress directly on reliability. Proposals like Reliability Banking [52] model reliability as a function of thermal history. The intuition here is that wearout is a cumulative process whose rate is strongly influenced by temperature. By maintaining a history of on-die temperatures these techniques can exploit the fact that applications are often comprised of hot and cool phases. Performance can be improved by accelerating (frequency scaling) periods of cool execution or by selectively disabling thermal throttling during periods of hot execution. Reliability targets are met as long as the cumulative thermal history for a processor is not allowed to exceed a predetermined threshold. Although this could be effective for truly homogeneous systems, in future CMPs with significant amounts of process variation, thermal stress is only half of the story. A comprehensive approach must also account for the existing damage already present in each core. More recent work by

Tiwari [105] addresses this shortcoming by monitoring the consumption of timing margins, implicitly accounting for both process variation and wearout. However, their main focus was on the use of Adaptive Body Biasing (ABB) and Adaptive Supply Voltage (ASV) scaling to increase lifetime operating frequencies. Their discussion of an aging-driven scheduling scheme is cursory and lacks the analysis and algorithmic tradeoffs presented in this work. Additionally, apart from relying heavily on static analysis of projected operating conditions, the proposal in [105] is also intrusive, requiring extensive hardware support (area and complexity) for ABB and ASV.

To summarize, previous perf-centric and thermal-p schedulers have illustrated how effective job scheduling can improve performance and power consumption. However, by not accounting for issues like the impact of process variation, module-level thermal variation across applications, system utilization, etc., existing thermal-r schedulers have not tapped the full potential of wearout-centric scheduling. The next section provides the motivation for this claim.

3.2.3 Workload Variation

Since both TDDB and NBTI are highly dependent on temperature, it is important to understand the thermal footprints of typical applications in order to appreciate the potential for reliability-centric scheduling. Figure 3.1 shows the range of temperatures experienced by different structures within an Alpha21364-like processor [5] across a set of 8 SPECINT (*bzip2, gcc, gzip, mcf, perlbmk, twolf, vortex, vpr*) and 9 SPECFP benchmarks (*ammp, applu, apsi, art, equake, galgel, lucas, sixtrack, swim, wupwise*). All temperatures are

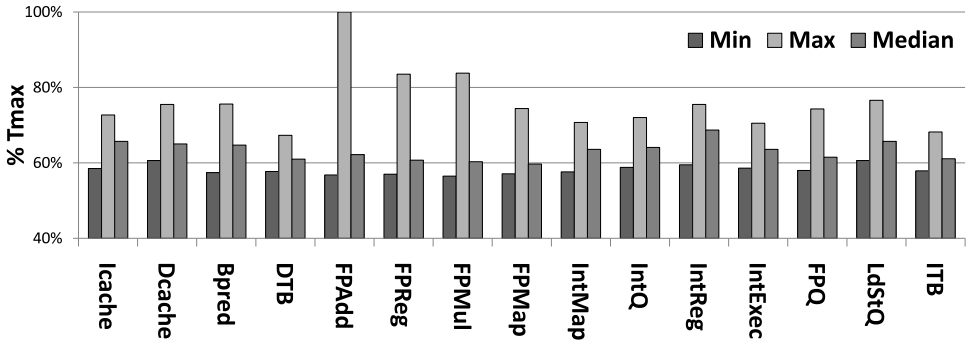
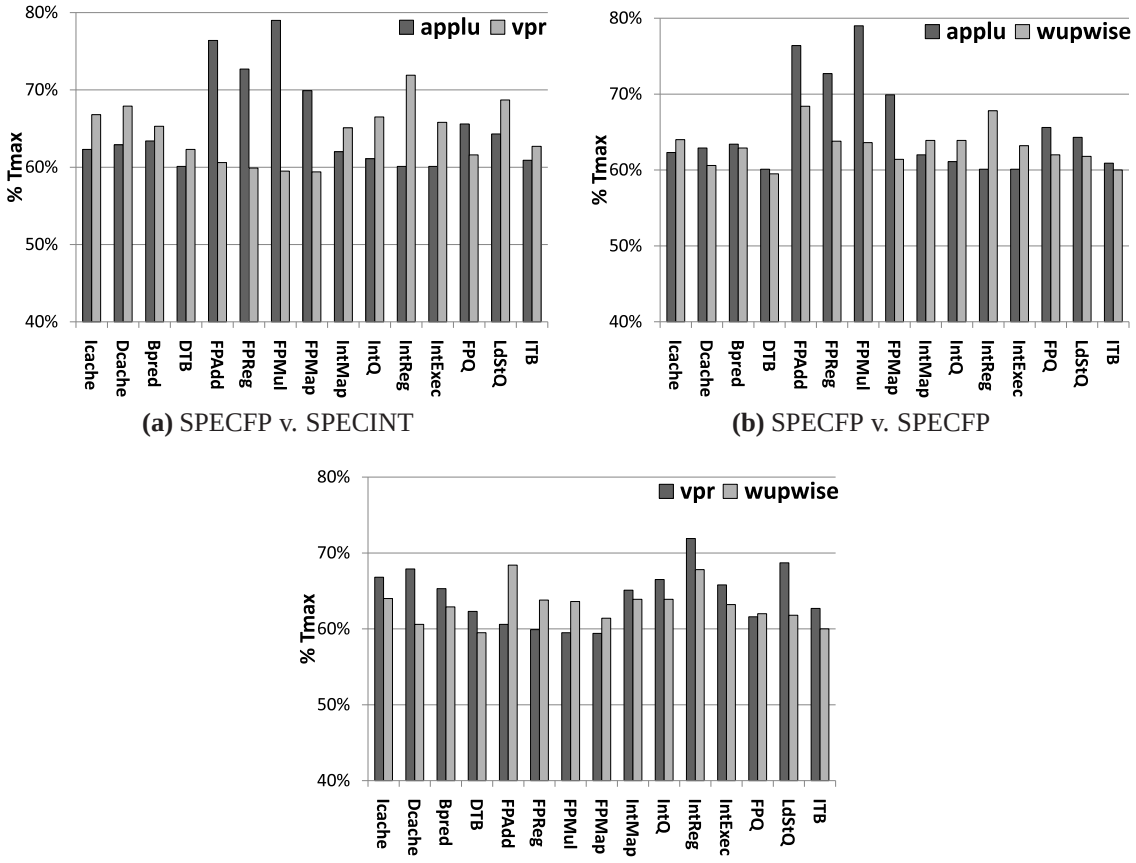


Figure 3.1: *Variation of module temperatures across SPEC2000 workloads. All temperatures are normalized to T_{max}, the peak temperature seen across all benchmarks and modules (83°C).*

normalized to the peak temperature, T_{max}, seen across all modules and benchmarks, which corresponds to the temperature of the FPAdd module when running *lucas* (83°C). Notice the significant variation in temperature within nearly every module. Apart from the more than 40% variation seen in FPAdd (a 37°C swing), other structures (whose utilizations are not as strongly correlated with the execution of floating-point and integer benchmarks) also exhibit significant temperature shifts, 10-15% for Bpred and IntReg. These large temperature ranges suggest that scheduling alone can be a powerful tool for manipulating aging rates.

Figure 3.2 selects a few representative applications and examines them in greater detail. Figures 3.2a and 3.2b highlight how the traditional view of "hot" and "cold" applications is perhaps too simplistic. Without accounting for the module-level variation in temperatures, one could incorrectly assume that *applu* is more taxing, from a reliability perspective, than *vpr* or *wupwise* simply because it exhibits a higher peak operating temperature (FPMul). However, this would neglect the fact that for many structures, like IntReg, temperatures for *applu* are actually much lower than the other two applications. For completeness, Figure 3.2c is included to show that variations in module temperatures exist even between



(a) SPECFP v. SPECINT

(b) SPECFP v. SPECFP

(c) Variation despite comparable peak temperatures

Figure 3.2: *Head-to-head comparisons of applu (SPECFP), vpr (SPECINT), and wupwise (SPECFP). No one benchmark in (a), (b), or (c) strictly dominates the other (with respect to temperature) across all modules.*

applications with comparable peak temperatures. All things considered, deciding where on the CMP to schedule a particular application, to achieve the least reliability impact, requires additional information about the strength of individual structures within every core. Although the magnitude of the temperature differences may not seem impressive at first, with peak deltas in module temperatures around 10-20% in Figure 3.2a, these modest variations in temperature can have dramatic impacts on a processor's mean time to failure (MTTF).

3.2.4 Implications for Mean Time to Failure

From Figure 3.2, one could expect a core consistently running *applu* to fail because of a fault in the `FPMul` unit due to its high operating temperatures. However, in the presence of process variation other structures within the core could have been manufactured with more defects (or tighter timing margins), and therefore even more susceptible to failure despite not ever realizing the same peak temperatures as `FPMul`. In this environment, a reliability-centric job scheduler must take into consideration the extent of damage present within a core in addition to the per-module thermal footprint of running applications. Figure 3.3 presents the expected lifetime of a core running *applu* or *vpr* as a function of the module identified as the weakest structure. The lifetimes are projected based on well-known MTTF equations for NBTI and TDDB [49, 97]. The values are normalized to the best achievable MTTF, which in this comparison is attained if `FPMap` is the weakest module in the core and the core is running *vpr*. The optimal job to schedule on a particular core to maximize its lifetime is dependent not just on the application mix currently available, but also on the strengths of individual structures within that core. Scheduling *applu* on a core with a weak `IntReg` can nearly triple its operating lifetime compared to naively forcing it to run *vpr*. Similarly, scheduling *vpr* instead of *applu* on a core with a weak `FPAdd` improves its projected lifetime by more than 4x.

To further highlight the need to address process and workload variation, a quick examination of the processors simulated in Section 3.4.2 reveals that 35% of core failures are the result of failing structures that never experience peak on-chip temperatures. Furthermore, 22% of core failures are caused by modules that do not rank among the top three

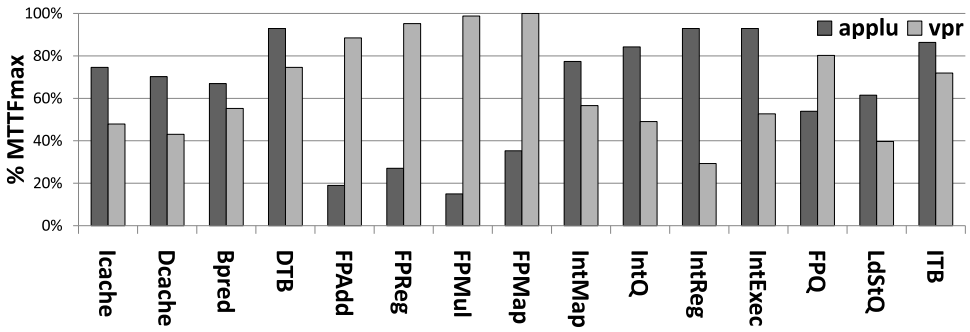


Figure 3.3: *Projected core lifetime based on execution of* applu *and* vpr *as a function of the module identified as the weakest structure. Values are normalized to the best achievable MTTF.*

most thermally active. By accounting for the impact of process variation and module-level thermal variation of applications, Maestro can prevent premature core failures and reap the opportunity left on the table by previous schedulers.

3.3 System Design

Figure 3.4 presents a block diagram of Maestro, which consists of two main components: 1) a health monitoring system (introspection) and 2) a virtualization layer that implements wearout-centric job scheduling (management). Although this chapter targets reliability-centric scheduling, a broader vision of introspective reliability management could use online sensor feedback to guide a range of solutions from traditional DVFS to more radical approaches like system-level reconfiguration [79, 39].

3.3.1 Health Monitoring

Tracking the evolution of wearout damage within a CMP (i.e., health monitoring) is essential to forming intelligent reliability-centric schedules. Maestro assumes that the underlying CMP is provisioned with circuit-level sensors like those described in [101]. Rec-

ognizing that the two mechanisms addressed in this work, NBTI and TDDB, both impact physical device parameters as they evolve has led researchers to actively develop circuit-level sensors that can track these changes. NBTI is known to shift threshold voltage (V_t) leading to slower devices and increased subthreshold/standby leakage current (I_{ddq}), while TDDB increases gate currents (I_{gs} and I_{gd}). Both result in statistically measurable degradation in timing paths at the microarchitectural-level [42, 102, 16, 22].

A runtime system collects raw data streams from the array of circuit-level sensors and applies statistical filtering and trend analysis (similar to what is described in [16]) to convert these streams into descriptions of system characteristics including, delay profiles, leakage currents, and operating temperatures. These individual channels of information are then processed to generate a comprehensive microarchitectural-level reliability assessment of the CMP. This is shown in Figure 3.4 as a vector of per-module damage values (relative to the maximum damage sustainable prior to failure). Introducing the additional analysis step allows the health monitoring system to account for things like the presence of redundant devices within a structure, the influence of shifting environmental conditions on sensor readings, and the interaction between different wearout mechanisms. Ultimately, this allows the low-level sensor feedback to be abstracted with each vector representing the effective damage profile for a particular core.

3.3.2 Maestro Virtualization Layer

The second portion of the Maestro framework resides in system firmware that serves as the interface between the OS and the underlying hardware. The OS provides the virtualization layer with a set of jobs that need to run on the CMP and other meta-data (optional)

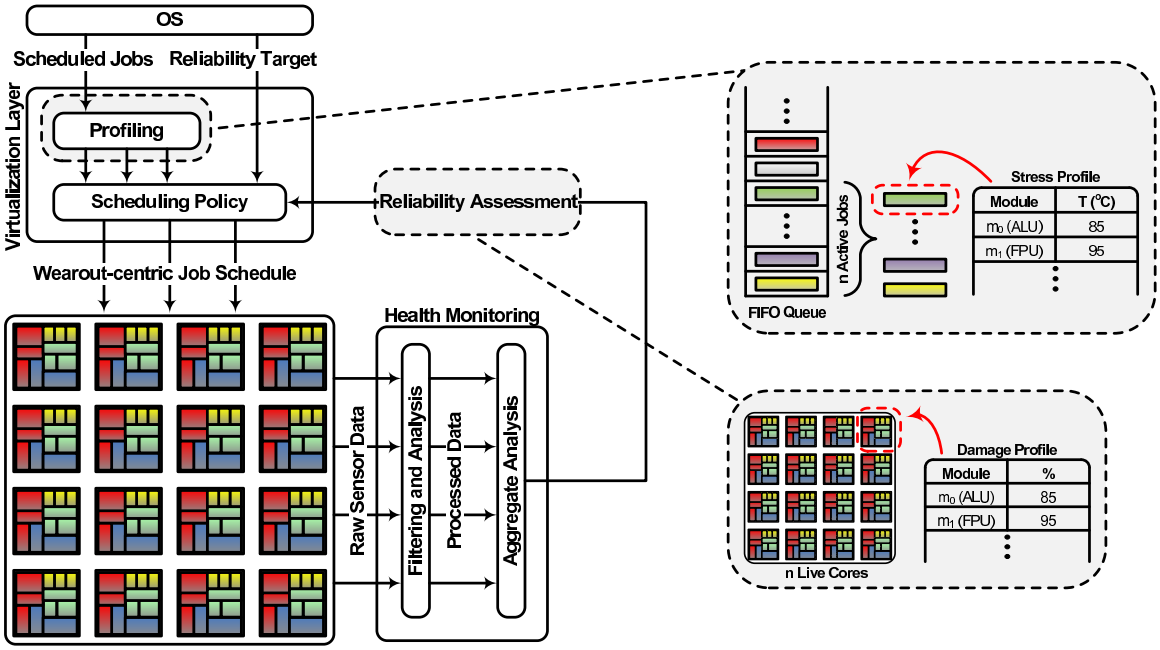


Figure 3.4: *A high-level block diagram of the Maestro introspective reliability management system. Dynamic monitoring of sensor feedback and detailed characterization of workload behavior enables Maestro to improve lifetime system reliability with wearout-centric scheduling.*

that can guide Maestro in refining its scheduling policies (Section 3.3.2.3). Online profiling of system workloads identifies application-specific thermal footprints, shown in Figure 3.4 as a vector of per-module temperatures for each application. This thermal footprint can either be generated by brief exploratory execution of jobs on the available cores, similar to what is done in [116], or projected by correlating thermal behavior with program phases (leveraging the existing body of work on runtime phase monitoring and prediction [40]). Given the prevalence of on-chip temperature sensors [36], Maestro assumes low-overhead exploration is performed during each scheduling interval. Coupled with the real-time health assessments, this detailed module-level application characterization enables Maestro to create wearout-centric job schedules that intelligently manage CMP aging.

As previously defined, scheduling in this chapter will refer to the act of mapping threads to cores and is initiated by two main events, 1) the OS issues new jobs for Maestro to ex-

ecute (pushes into a FIFO queue) or 2) the damage profile of the underlying CMP has changed sufficiently (taking on the order of days/weeks) to warrant thread migration. The two reliability-centric scheduling policies evaluated in this work illustrate two approaches to lifetime reliability. The greedy policy (Section 3.3.2.2) takes the position that all core failures are unacceptable and aggressively preserves even the weakest cores. The adaptive policy (Section 3.3.2.3) champions a more unconventional philosophy that claims individual core failures are tolerable provided the lifetime reliability of the CMP system is maximized.

Both wearout-centric policies, and the naive baseline scheduler, are presented below along with corresponding pseudocode. Unless otherwise indicated, the following definitions are common to all policies:

m: a microarchitectural module (i.e., `FPMul`, `IntReg`, etc.).

LiveCores: the set of functional cores in the CMP, $\{c_0, c_1, ..., c_N\}$.

JobQueue: the set of **all** pending, uncompleted jobs issued from the OS.

ActiveJobs: the set of the N oldest, uncompleted, jobs, $\{j_0, j_1, ..., j_N\}$.

Dmg(m): the entry in the CMP damage profile for module m.

Temp(j, m): the entry for module m in the temperature footprint for job j.

3.3.2.1 Naive Scheduler

A standard round-robin scheduler is used as the baseline policy. The least-recently-used (LRU) core in the set of $LiveCores$ is assigned the oldest job from the set of $ActiveJobs$.

Algorithm 1 Greedy wearout-centric scheduler

Step 1:
  foreach $c \in LiveCores$ **do**
    find c_{dmg} , the damage present in core c , **where**
      $c_{dmg} \longleftarrow Dmg(m') \mid m' \in c \wedge Dmg(m') \geq Dmg(m), \forall m \in c$
  end
  sort $LiveCores$ based on c_{dmg}
end
Step 2:
  until $ActiveJobs$ is empty
    $c_w \longleftarrow$ weakest core in $LiveCores$ based on c_{dmg}
    $m_w \longleftarrow m' \mid m' \in c_w \wedge Dmg(m') \geq Dmg(m), \forall m \in c_w$
    foreach $j \in ActiveJobs$ **do**
      find $cost_{j,c_w}$, the cost of executing job j on core c_w , **where**
        $cost_{j,c_w} \longleftarrow Temp(j, m_w)$
    end
    $j_{opt} \longleftarrow j' \mid j' \in ActiveJobs \wedge cost_{j',c_w} \leq cost_{j,c_w}, \forall j \in ActiveJobs$
    Assign job j_{opt} to core c_w
    Remove c_w from $LiveCores$ and j_{opt} from $ActiveJobs$
  end
end

This process is repeated until all jobs in $ActiveJobs$ have been scheduled. This policy maintains high-level load balancing by distributing jobs uniformly across the cores. However, without accounting for core damage profiles or application thermal footprints, the resulting schedule is effectively a random mapping (from a reliability perspective).

3.3.2.2 Greedy Scheduler

This policy attempts to minimize the number of premature core failures by greedily favoring the weakest cores (Algorithm 1). Cores are sorted based upon their damage profiles and priority is given to the cores whose weakest modules possess the most damage (Step 1 of Algorithm 1). These "weak" cores are greedily assigned jobs with the most favorable thermal footprints with respect to their damage profiles (Step 2 of Algorithm 1),

Algorithm 2 Adaptive wearout-centric scheduler

let $GA(J, C)$ be the optimal schedule generated by the GA for jobs J and cores C
Step 1:
  foreach $c \in LiveCores$ **do**
    find c_{dmg} , the damage present in core c , **where**
      $c_{dmg} \longleftarrow \sum_{m_i}^{c} \alpha_i Dmg(m_i)$ and α_i is a scaling factor biased toward modules with more damage
  end
  sort $LiveCores$ in increasing order of c_{dmg}
  $PrimaryCores \longleftarrow$ first n cores **where** n is set by the user through the OS
  $SecondaryCores \longleftarrow$ remaining $N - n$ cores
end
Step 2:
  let $S_{primary}$, be the set of job-to-core assignments, (j, c), $\forall c \in PrimaryCores$
  $S_{primary} \longleftarrow GA(ActiveJobs, PrimaryCores)$
  Assign jobs for $PrimaryCores$ according to $S_{primary}$
  Remove assigned jobs from $ActiveJobs$
end
Step 3:
  let $S_{secondary}$, be the set of job-to-core assignments, (j, c), $\forall c \in SecondaryCores$
  $S_{secondary} \longleftarrow GA(ActiveJobs, SecondaryCores)$
  Assign jobs for $SecondaryCores$ according to $S_{secondary}$
end

minimizing their effective thermal stress. This *local wear-leveling* reduces the probability that these weak cores will fail due to a *single* damaged structure. Scheduling the weak cores first maximizes the probability of finding jobs with favorable thermal footprints with respect to each weak core since there is a larger application mix to choose from. However, this also forces the stronger cores to execute the remaining, potentially less desirable, jobs. In practice, this means that the stronger cores in the CMP actually sacrifice a portion of their lifetime to lighten the burden on their weaker counterparts (*global wear-leveling*).

3.3.2.3 Adaptive Scheduler

The adaptive scheduler recognizes that many CMP systems are often underutilized, provisioned with more cores than they typically have jobs to run (see Section 3.4.4). The

scheduler exploits this fact by allowing a few weak cores to be sacrificed in order to preserve the remaining stronger cores (Algorithm 2). Although being complicit in core failures may seem non-intuitive, in systems that are underutilized, the greedy scheduler can lead to CMPs that are overprovisioned early in the CMP's life ($LiveCores >> JobQueue$) while not assuring enough available throughput ($LiveCores < JobQueue$) later on. This insight forms the basis of the adaptive policy.

Promoting a survival-of-the-fittest environment, this policy maximizes the functional life of the strongest subset of cores ($PrimaryCores$ in Step 1 of Algorithm 2), those with the least amount of initial damage and the potential to have the longest lifetimes. By assigning jobs to the $PrimaryCores$ first, Maestro ensures that they execute applications with the most appropriate thermal footprints (Step 2 of Algorithm 2). The remaining jobs are assigned amongst the $SecondaryCores$ (Step 3 of Algorithm 2). This can lead to some weak cores failing sooner than under a greedy policy. Note, however, in Step 3 of Algorithm 2, the scheduler is still looking amongst the remaining jobs for the one with the best thermal footprint given a core's damage profile. This *local wear-leveling*, common to both the greedy and adaptive policies, ensures that the weaker cores even under the adaptive policy survive longer than they would under the naive policy. Ultimately, over the lifetime of the CMP, if $PrimaryCores \geq JobQueue$ consistently, while avoiding periods when $PrimaryCores >> JobQueue$ or $PrimaryCores < JobQueue$, then Maestro has maximized the total amount of computation performed by the system. The proper size of $PrimaryCores$, n, is exposed to the OS so that the behavior of the scheduler can be customized to the needs of the end user.

Finally, note in Step 2 and Step 3 of Algorithm 2, the scheduler uses an optimization scheme based on a genetic algorithm (GA) to identify the least-cost schedules for both the $PrimaryCores$ and $SecondaryCores$. This allows the adaptive scheduler to consider the effect scheduling a job has on all structures within a core (unlike the greedy scheduler which only looks at the weakest structure) for more effective *local wear-leveling*.

3.3.2.4 GA optimization

The optimization used in this work is derived from [25], a standard solution of the generalized assignment problem, and is described below [1].

Chromosome definition: The chromosome modeled is a job-to-core mapping of a set of n jobs, $J = \{j_0, j_1, ..., j_n\}$, to a set of m cores $C = \{c_0, c_1, ..., c_n\}$. It is represented as a one-dimensional array where the value stored at index i, j_i, is the job that has been assigned to core i. The example in Figure 3.5 has jobs j_1 mapped to core 0, j_{n-1} mapped to core 1, and j_0 mapped to core m. During **Step 2** of the adaptive scheduling algorithm $n > m$, while for the optimization performed in **Step 3** $m = n$.

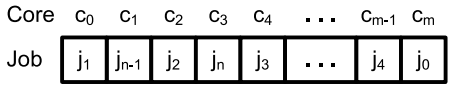


Figure 3.5: *Chromosome structure*

Cost function: The cost function used by the GA is recalculated at each scheduling interval, based on the CMP damage profile and application thermal footprints, according to

[1]The runtime overhead of the GA is negligible for long-running scientific and server workloads. However, for shorter-running applications the GA optimization can be replaced by a greedy version without severely impacting the effectiveness of the adaptive scheduler.

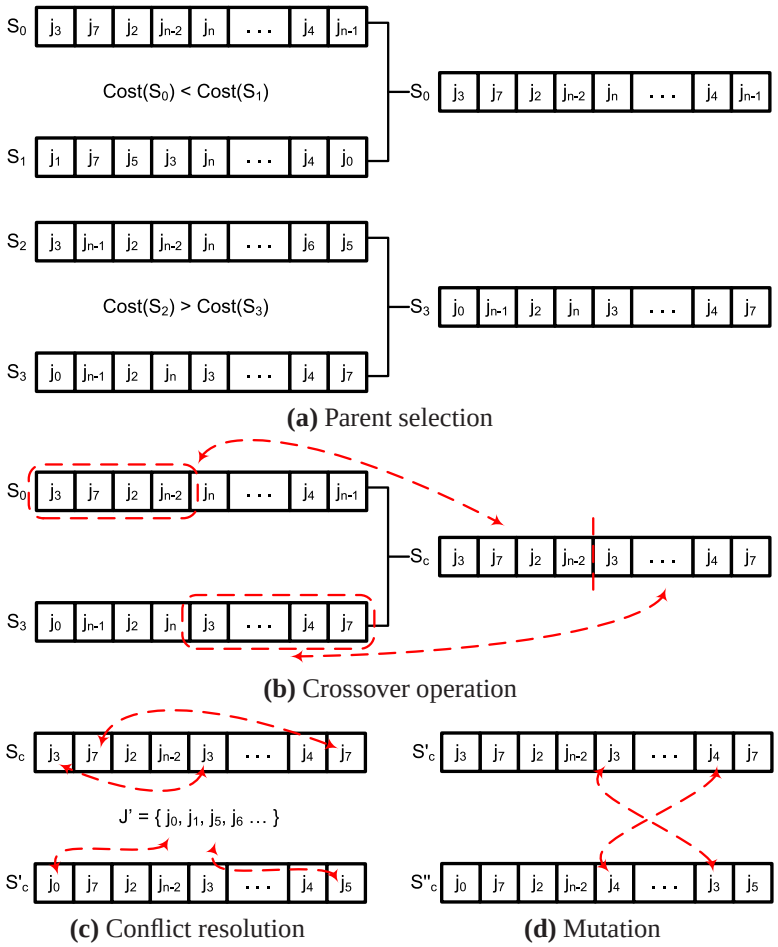


Figure 3.6: *Steps involved in reproduction.* S_0, S_1, S_2, S_3 *are the parental candidates.* S_c *is the resulting child chromosome after initial crossover.* S'_c *and* S''_c *are the states of the child chromosome after conflicts resolution and mutation respectively.*

Equation 3.1, where $Cost(S)$ = the cost of schedule S and $Cost(j, c)$ = the cost of scheduling job j on core c.

$$\begin{aligned} Cost(S) &= \sum_{j,c}^{S} Cost(j,c) \\ &= \sum_{j,c}^{S} \Big(\sum_{m}^{c} Dmg(m) \cdot Temp(j,m) \Big) \end{aligned} \tag{3.1}$$

The individual steps of the GA are enumerated below:

1. **Generate initial population:** An initial population of solutions (schedules) is created by randomly enumerating a subset of the possible job-to-core mappings.

2. **Evaluate fitness:** Calculate the fitness (cost) of all members of the population using Equation 3.1.

3. **Reproduction:** Two parents are identified, each using a simple binary tournament where two candidates are selected randomly from the population and the one with the best fitness (smallest cost) is chosen for reproduction (Figure 3.6a). A child is generated by applying a one-point crossover operator on the parent chromosomes, where a random crossover point $i \in [0, m]$ is selected, where m is the size of the chromosomes. The child chromosome is formed by combining the first i genes from one parent with the last $m - i$ genes from the second parent (Figure 3.6b) . Note that this newly formed chromosome could have the same job assigned to two different cores. For this case to arise there must also be a set of jobs $J' \subset J$ that are unassigned since $n \geq m$. To resolve the conflicts, one of the redundant cores (selected at random) is reassigned a job from J' based on $Cost(j, c)$ (Figure 3.6c). Lastly, the newly formed child chromosome is mutated by taking 2 randomly selected job assignments and swapping them (no risk of creating new conflicts), reducing the probability of converging at local optima (Figure 3.6d) .

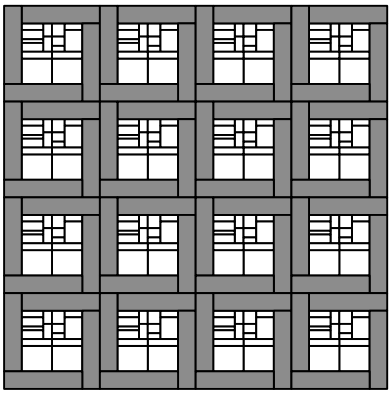
4. **Replace and Repeat:** After a child solution is formed the weakest member, as defined by the cost function, of the existing population is replaced by the new child. This concludes a single generation in the evolutionary cycle. The process is repeated

until a predetermined number of generations, gen_{max}, fails to produce an improved solution[2].

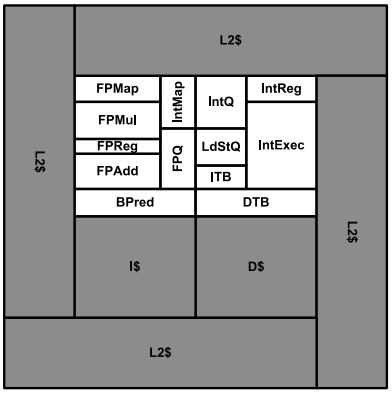
3.4 Evaluation and Analysis

This section evaluates Maestro's reliability-centric scheduling policies using lifetime reliability simulations. A variety of system parameters including CMP size and system utilization are varied to investigate their impact on Maestro's performance. The effectiveness of each wearout-centric policy is measured in terms of *lifetime throughput* (LT), the number of cycles spent executing active jobs (real applications not idle threads), summed across all cores, throughout the entire lifetime of the CMP. LT improvement metrics are the result of comparisons with the naive, round-robin scheduler presented in Section 3.3.2.1. Monte Carlo experiments are conducted using a simulation setup similar to the framework in [35]. The standard toolchain of SimpleScalar [10], Wattch [21], and Hotspot [88] is used to simulate the thermal characteristics of workloads and Varius [82] is used to model the impact of process variation. Results presented in this section, unless otherwise indicated, are for a 16-core CMP with processors modeled after the DEC Alpha 21264/21364 [5]. Details of the CMP configuration can be found in Figure 3.7.

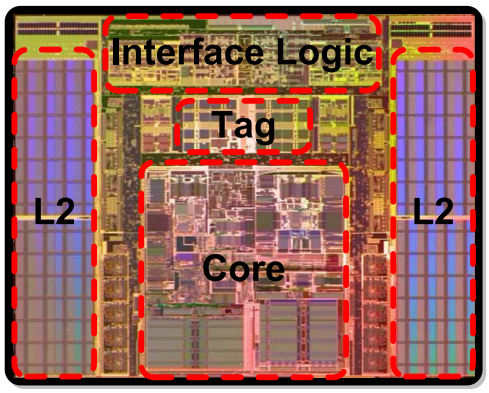
[2]Given the size of the solution space, as many as $16!$ possible schedules for a 16-core CMP, values of gen_{max} from 0 to 100,000 were studied to understand the tradeoff between optimality and runtime. The actual values of gen_{max} used in Section 3.4 were determined empirically based on the CMP size, with many runs producing good results with gen_{max} as low as 1000.



(a) CMP floorplan



(b) Core floorplan



(c) Alpha 21364 die photo

Core (Alpha 21264/21364 – like)

Fetch	4 wide
INT Register File	80-entry
INT Issue	4 wide
INT Functional Units	4 ALUs, 4 mult/div (2 clusters)
FP Register File	80-entry
FP Issue	2 wide
FP Functional Units	1 adder, 1 mult/div (1 cluster)

Memory

LdQ/StQ	32-entries each
L1 (I$ and D$)	Private 2-way, 64K, 2-cycle access
L2	Private 1M, 8-way, 10-cycle access

Process

Technology	32nm (96mm^2 CMP)
Frequency	4 GHz
Vdd	1.0 V

(d) Core specifications

Figure 3.7: *CMP details. All simulation results, unless otherwise stated, are presented for a CMP configured with 16 cores.*

3.4.1 Adaptive Lifetime Simulation

Given that CMPs have lifespans on the order of years (3-5 years in future computer systems [33]), detailed lifetime reliability simulation is a computationally intensive task. This is especially true when large numbers of Monte Carlo runs must be conducted to generate statistically significant results. Since wearout damage takes years to reach critical mass, results presented in this section were gathered using an *adaptive* simulation scheme. Short periods of detailed system-level reliability simulation, the darker phases in Figure 3.8a, are used to gather statistics on the progression of CMP aging in light of dynamically changing workload streams and Maestro's reliability-centric scheduling. The simulation is then

rapidly advanced through a longer period of time (accelerated simulation) using the statistics generated during the most recent detailed phase as a guide. To minimize error, the length of the accelerated simulation phase is limited by the amount of damage accumulated during the detailed interval according to Equation 3.2:

$$L_a = (\frac{D_{fail}}{D_{acc}}) \cdot AF \cdot L_d \text{ where,} \tag{3.2}$$

L_a = the length of the accelerated phase.

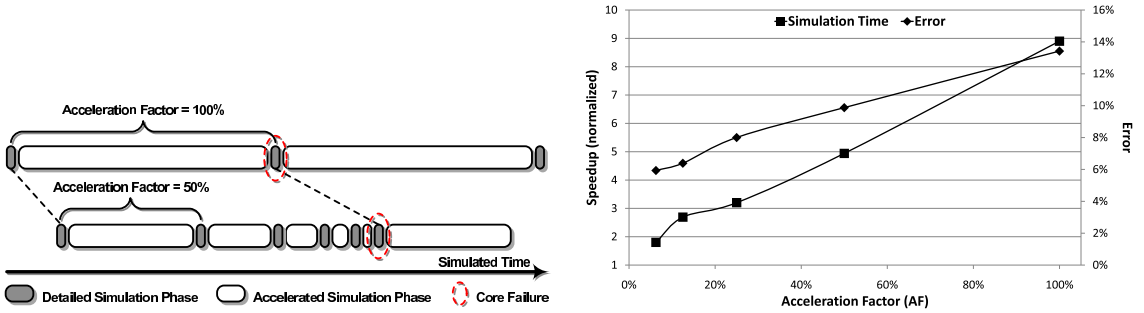
L_d = the length of the previous detailed phase.

D_{fail} = the amount of damage the weakest core in the CMP can sustain before failing.

D_{acc} = the amount of damage accumulated by the weakest core during the previous detailed phase.

AF = variable parameter that trades off simulation time for accuracy (0%-100%).

Dynamically adjusting the durations allows simulation to slow down as cores near their failing point, where small changes in damage and scheduling decisions have larger implications. When a core fails in phase i, accelerated simulation resumes at a faster rate ($L_{a_{i+1}} > L_{a_i}$), but L_a soon contracts as the next core in the CMP nears failure. Figure 3.8a illustrates (not to scale) how adjusting AF can influence the lengths of the accelerated phases. The value of AF essentially dictates the number of detailed phases that are simulated be-



(a) Interleaving of detailed and accelerated simulation phases.

(b) Simulation time/error v. acceleration factor (AF).

Figure 3.8: *The adaptive simulation used to accelerate lifetime reliability simulations while incurring minimal experimental error.*

tween core failures. At an AF of 100%, simulations are accelerated from one core failure to the next. However, when AF is dialed down to 50%, many more phases are required to cover the same amount of simulated time, concentrating simulation effort around times when cores are failing and improving simulation accuracy. Figure 3.8b shows both simulation time speedup and error as a function of AF, illustrating how simulation time can be traded off for fidelity. The experiments presented in this work use an AF of 6%, resulting in simulation runtimes from 30 minutes to over 6 hours for a single Monte Carlo run.

3.4.2 Lifetime Throughput Enhancement

Figure 3.9 shows the normalized LT improvement as a function of the scheduling policy, CMP size, and failure threshold. In the context of this chapter, failure threshold is defined as the number of cores that must fail before a chip is considered unusable. This is the point at which the risks/costs associated with maintaining a system with only a fraction of its original computational capacity justifies replacing the chip. The CMP is considered dead even though functional cores still remain. The results shown in Figure 3.9 are conducted for 2 to 16-core systems, and failure thresholds ranging from 1 core to all cores. The

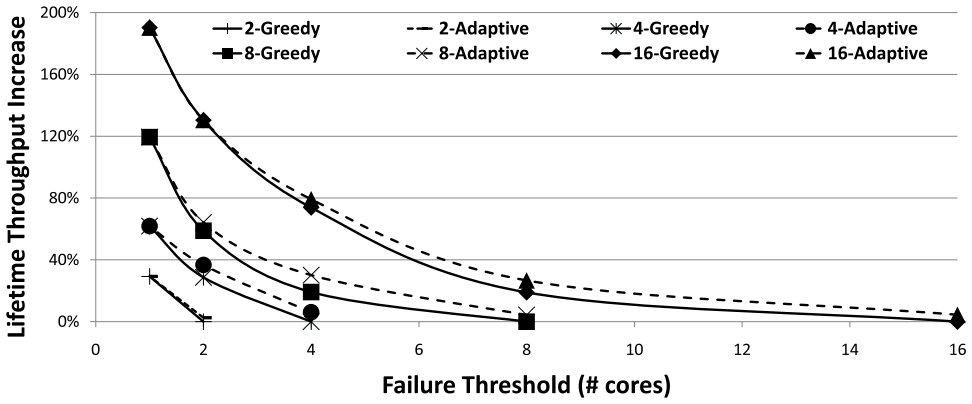


Figure 3.9: *Performance of wearout-centric scheduling policies verses CMP size and failure threshold.*

value of the failure threshold is passed to the adaptive policy so that it can optimize for the appropriate number of cores. Results are shown for CMP utilizations of 100%, providing a lower-bound on the benefits of the adaptive policy (Section 3.4.4 examines the impact of CMP utilization).

As expected, both the greedy and adaptive policies perform well across all CMP sizes and the majority of failure thresholds. As the size of the CMP grows, Maestro has more cores to work with, increasing the chances of finding complementary job-to-core mappings. This results in more effective schedules for both wearout-centric policies improving their performance. Yet even with the lack of scheduling alternatives in a 2-core system, both policies can still achieve a respectable 30% improvement.

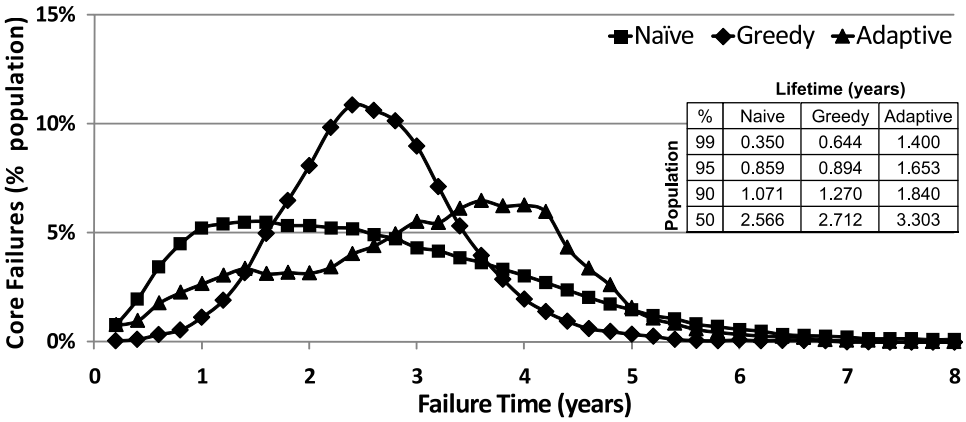
A strong dependence on failure threshold is also evident. By aggressively minimizing premature core failures, the greedy scheduler achieves large gains for small failure thresholds. However, as the failure threshold nears the size of the CMP, the LT improvement attenuates. This is expected since under the greedy policy, stronger cores sacrifice a portion of their lifetime in order to preserve their weaker counterparts. The cost of this

sacrifice is most apparent when the failure threshold allows all the cores to fail. In these systems, the increased contribution toward LT by the weak cores is offset by the loss in LT resulting from the strong cores failing earlier. Notice also that the adaptive scheduler outperforms greedy by the largest margins when the failure threshold is roughly half the size of the CMP. In these situations, the adaptive scheduler has the maximum freedom to sacrifice $SecondaryCores$ to preserve $PrimaryCores$ (Section 3.3.2.3). At either extreme for failure threshold, it performs similarly to greedy.

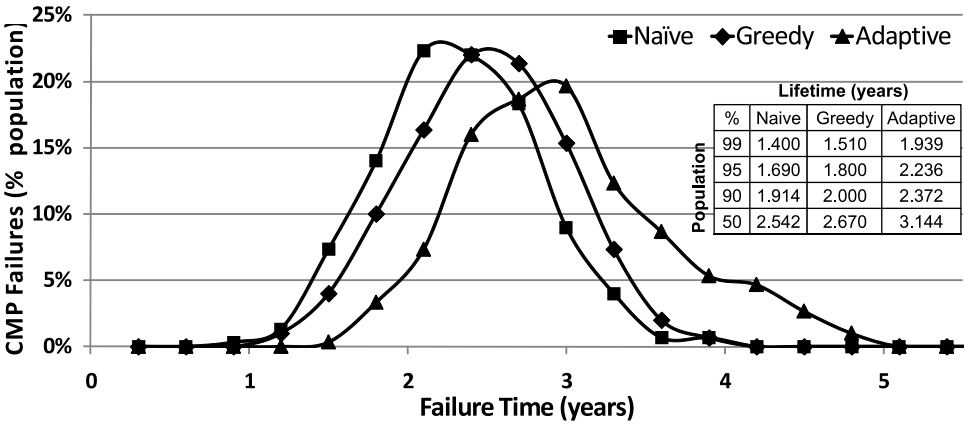
Lastly, it is important to note that, although the benefits of wearout-centric scheduling are less impressive for these extreme values of failure threshold, the scenarios when a user could actually afford to wait for all the cores within a system to fail are also quite remote. For the remainder of the chapter, all the experiments shown are for a 16-core CMP with a failure threshold of 8 cores and 100% system utilization unless otherwise indicated.

3.4.3 Failure Distributions

Figure 3.10 presents the failure distributions for the individual cores, as well as the CMPs that correspond to the results in Figure 3.9. Figure 3.10a illustrates the effectiveness of the wearout-centric policies at distributing the workload stress appropriately. The distribution for the baseline naive policy reveals a bias towards early premature core failures. The greedy scheduler, exploiting effective wear-leveling, produced a tighter distribution, lacking in both premature failures as well as cores that significantly outlasted their peers. Lastly, the adaptive policy also delivers on its promises by preserving a subset of cores for a longer period of time than either the naive or greedy schedulers.



(a) Failure distribution (Core)



(b) CMP failure distribution (CMP)

Figure 3.10: *Failure distributions for individual cores and the 16-core CMP with a failure threshold of 8 cores and 100% utilization. Trendlines are added (between markers) to improve readability.*

Figure 3.10b tells a similar story, but with chip-level failures. As with the individual core distributions, both wearout-centric policies are able to increase the mean failure time of the CMP population. Note that because the failure time of a CMP is limited by the weakest set of its constituent cores, the distributions in Figure 3.10b are considerably tighter than those in Figure 3.10a. The corresponding tables of expected lifetimes embedded within the plots present the data slightly differently. From a product yield/warranty perspective, intelligent wearout-centric scheduling can be thought of as an additional means of ensuring that cores meet their expected reliability qualified lifetimes. For example, the table in Fig-

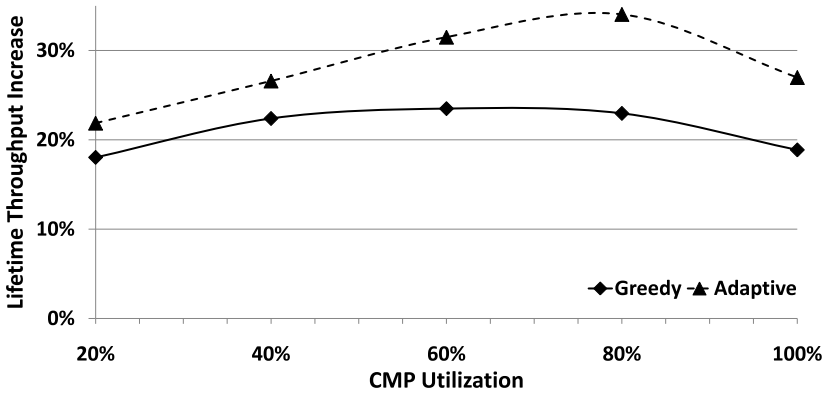


Figure 3.11: *Impact of CMP utilization on reliability enhancement.*

ure 3.10b shows that the adaptive scheduler enabled 99% of the chips to survive beyond 1.9 years, compared to just 1.4 years with the naive baseline, a 38% improvement. Granted, job assignment alone cannot make *guarantees* on lifetime, but it can complement existing more aggressive techniques like thermal throttling.

3.4.4 Sensitivity to System Utilization

The utilization of computer systems can be highly variable, both within the same domain (e.g., variability inside data centers) and across domains. One might expect computationally intensive scientific codes (e.g., physics simulations, oil exploration, etc.) to consistently utilize the hardware. On the other hand, since designers build web servers to accommodate peak loads (periodic by season, day, and hour), they are often over-provisioned for the common case. Some reports claim average utilization as low as 20% of peak [6].

Figure 3.11 plots the performance of Maestro's wearout-centric schedulers as a function of system utilization. The results are shown for nominal utilizations ranging from 20% (light duty mail server or embedded system) to 100% (scientific cluster)[3]. Note that ini-

[3]Although the mean utilization per simulation run is fixed, the instantaneous utilization experienced by the CMP is allowed to vary over time, sometimes peaking at 100% even for a system

tially as average utilization drops, improvement in lifetime throughput actually increases. A system that is slightly underutilized can be more aggressively load balanced since some cores are allowed to remain idle. However, as utilization continues to drop these gains are eventually lost, until finally improvements are actually worse than at full utilization. In these highly over-provisioned systems, the efforts of wearout-centric scheduling to prevent premature failures are *partially* wasted because so few cores are actually necessary to sustain demand. Nevertheless, in the long run, the periodic spikes in utilization do accumulate, and thanks to the longer overall core lifetimes (lower utilization means less overall stress that translates to longer lifetimes), the greedy and adaptive schedulers still manage to exhibit improvements.

3.4.5 Sensitivity to Sensor Noise

Figure 3.12 illustrates how error-prone sensors could impact lifetime reliability gains. Although the introduction of systematic error, which is studied in Figure 3.12b, does reduce the potential of wearout-centric scheduling, the presence of random noise (more common for circuit-level sensors) shown in Figure 3.12a can be accounted for and mitigated by the statistical filtering and trend analysis schemes referenced in Section 3.3.1. Yet, even at the extreme of +/-15% systematic error, Maestro still achieves over 10% LT improvement. Figure 3.12b also suggests that the adaptive scheduler is more sensitive to noise than the greedy scheduler. By aggressively trying to preserve $PrimaryCores$, the adaptive heuristic relies strongly on sensor feedback to accurately identify the boundary between its two classes of processors, making it less robust against sensor inaccuracy.

nominally at 20% load. Furthermore, the average *effective* utilization is also changing as cores on the CMP begin to fail.

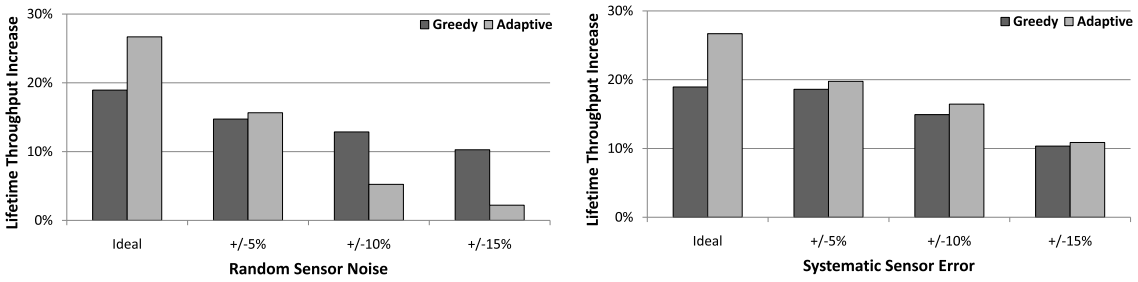


Figure 3.12: *Sensitivity to sensor noise. Although random sensor noise can be removed with the appropriate filtering, systematic error due to manufacturing tolerances is more problematic.*

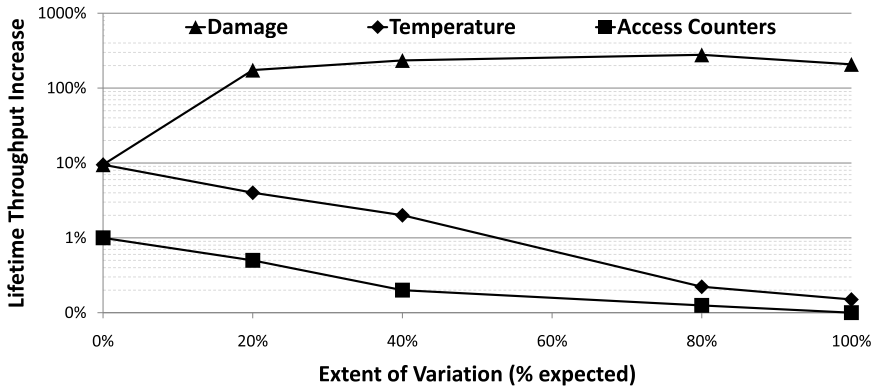


Figure 3.13: *Performance of wearout-centric scheduling with different sensors. Results are shown for a failure threshold of 1 core to favor the temperature sensor and access counter based approaches.*

3.4.6 Sensor Selection

Lastly, Figure 3.13 presents a comparison between the low-level damage sensors advocated in this work and more conventional hardware like temperature sensors and performance counters. Given that Maestro is targeting an environment with significant amounts of process variation, it is not surprising that employing temperature and activity readings as proxies for wearout/manufacturing induced damage is inadequate. They are unable to account for the extent to which non-uniform, pre-existing damage within the CMP responds to the same thermal stimuli. In the absence of variation, a scheduler relying on only temperature might effectively enhance lifetime reliability by evenly distributing the

thermal stress across the CMP. However, without any knowledge of CMP damage profiles, as process variation is swept from one extreme (no variation) to the other (100% expected variation at 32nm), thermal load balancing alone is insufficient and Figure 3.13 shows a dramatic plunge in the effectiveness of these temperature based schemes. Similarly, the performance counter approach performed poorly across the spectrum of variation.

3.5 Summary

As large CMP systems grow in popularity and technology scaling continues to exacerbate lifetime reliability challenges, the research community must develop innovative ways for systems to dynamically adapt. Although issues like process variation are the source of design and validation nightmares, this inherent heterogeneity in future systems is also a source of potential opportunity. Maestro recognizes that although emerging reliability obstacles cannot be ignored, with the appropriate monitoring and intelligent management, they can be overcome. By exploiting low-level sensor feedback, Maestro was able to demonstrate the effectiveness of wearout-centric scheduling at preventing premature core failures, improving expected CMP lifetimes by as much as 38%. Formulating wearout-centric schedules that achieved both local and global wear-leveling, Maestro enhanced the lifetime throughput of a 16-core CMP by as much as 180%. Future work that leverages sensor feedback to improve upon other traditional reliability management mechanisms (e.g., DVFS) could demonstrate still more potential.

CHAPTER IV

Shoestring: Probabilistic Soft Error Reliability on the Cheap

4.1 Introduction

A critical aspect of any computer system is its reliability. Computers are expected to perform tasks not only quickly, but also correctly. Whether they are trading stocks from a laptop or watching the latest YouTube video on an iPhone, users expect their experience to be fault-free. Although it is impossible to build a completely reliable system, hardware vendors target failure rates that are imperceptibly small.

One pervasive cause of computer system failure and the focus of this chapter is soft errors. A soft error, or transient fault, can be induced by electrical noise or high-energy particle strikes that result from cosmic radiation and chip packaging impurities. Unlike manufacturing or design defects, which are persistent, transient faults as their name suggests, only sporadically influence program execution.

One of the first reports of soft errors came in 1978 from Intel Corporation, when chip packaging modules were contaminated with uranium from a nearby mine [53]. In 2004,

Cypress semi-conductor reported a number of incidents arising from soft errors [121]. In one incident, a single soft error crashed an entire data center and in another soft errors caused a billion-dollar automotive factory to halt every month.

Since the susceptibility of devices to soft error events is directly related to their size and operating voltage, current scaling trends suggest that dramatic increases in microprocessor soft error rates (SER) are inevitable. Traditionally, reliability research has focused largely on the high-performance server market. Historically the gold standards in this space have been the IBM S/360 (now Z-series servers) [95] and the HP NonStop systems [14], which rely on large scale modular redundancy to provide fault tolerance. Other research has focused on providing fault protection using redundant multithreading [80, 74, 59, 38, 90] or hardware checkers like DIVA [114, 19]. In general, these techniques are expensive in terms of both the area and power required for redundant computation and are not applicable outside mission-critical domains.

The design constraints of computer systems for the commodity electronics market differ substantially from those in the high-end server domain. In this space, area and power are primary considerations. Consumers are not willing to pay the additional costs (in terms of hardware price, performance loss, or reduced battery lifetime) for the solutions adopted in the server space. At the same time, they do not demand "five-nines" of reliability, regularly tolerating dropped phone calls, glitches in video playback, and crashes of their desktop/laptop computers (commonly caused by software bugs). The key challenge facing the consumer electronics market in future deep submicron technologies is providing just enough coverage of soft errors, such that the effective fault rate (the raw SER scaled by the

available coverage) remains at level to which people have become accustomed. Examining how this coverage can be achieved “on the cheap” is the goal of this chapter.

To garner statistically high soft error coverage at low overheads, we propose Shoestring, a software-centric approach for detecting and correcting soft errors. Shoestring is built upon two areas of prior research: symptom-based fault detection and software-based instruction duplication. Symptom-based detection schemes recognize that applications often exhibit anomalous behavior (symptoms) in the presence of a transient fault [111, 48]. These symptoms can include memory access exceptions, mispredicted branches, and even cache misses. Although symptom-based detection is inexpensive, the amount of coverage that can be obtained from a symptom-only approach is typically limited. To address this limitation we leverage the second area of prior research, software-based instruction duplication [75, 76]. With this approach, instructions are duplicated and results are validated within a single thread of execution. This solution has the advantage of being purely software-based, requiring no specialized hardware, and can achieve nearly 100% coverage. However, the overheads in terms of performance and power are quite high since a large fraction of the application is replicated.

The key insight that Shoestring exploits is that the majority of transient faults can either be ignored (because they do not ultimately propagate to user-visible corruptions at the application level) or are easily covered by light-weight symptom-based detection. To address the remaining faults, compiler analysis is utilized to identify high-value portions of the application code that are both susceptible to soft errors (i.e., likely to corrupt system state) and statistically unlikely to be covered by the timely appearance of symptoms. These portions of the code are then protected with instruction duplication. In essence, Shoestring

intelligently selects between relying on symptoms and judiciously applying instruction duplication to optimize the coverage and performance tradeoff. In this manner, Shoestring transparently provides a low-cost, high-coverage solution for soft errors in processors targeted for the consumer electronics market. However, unlike the high-availability IBM and HP servers which can provide provable guarantees on coverage, Shoestring provides only opportunistic coverage, and is therefore not suitable for mission-critical applications.

The contributions of this chapter are as follows:

- A transparent software solution for addressing soft errors in commodity processors that incurs minimal performance overhead while providing high fault coverage.
- A new reliability-aware compiler analysis that quantifies the likelihood that a fault corrupting an instruction will be covered by symptom-based fault detection.
- A selective instruction duplication approach that leverages compiler analysis to identify and replicate a small subset of vulnerable instructions.
- Microarchitectural fault injection experiments to demonstrate the effectiveness of Shoestring in terms of fault coverage and performance overhead.

4.2 Background and Motivation

4.2.1 Soft Error Rate

The vulnerability of individual transistors to soft errors is continuing to grow as device dimensions shrink with each new technology generation. Traditionally, soft errors were a major concern for memory cells due to their higher sensitivity to changes in operating

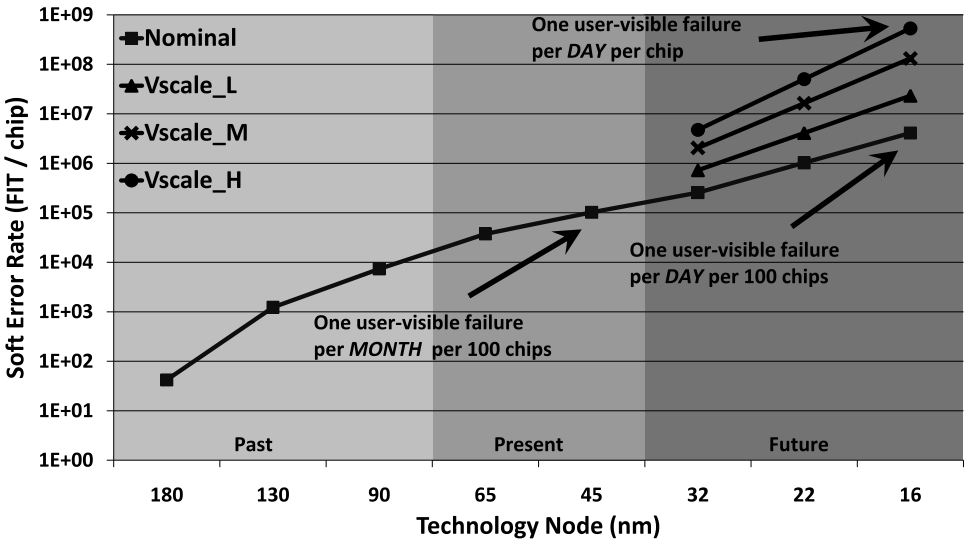


Figure 4.1: *The soft error rate trend for processor logic across a range of silicon technology nodes. The* Nominal *curve illustrates past and present trends while the* Vscale_L, Vscale_M, *and* Vscale_H *curves assume low, medium and high amounts (respectively) of voltage scaling in future deep submicron technologies. The user-visible failure rates highlighted at* $45\,nm$ *and* $16\,nm$ *are calculated assuming a 92% system-wide masking rate.*

conditions. However, protecting memory cells is relatively straightforward using parity checks or error correcting codes (ECC). On the other hand, combinational logic faults are harder to detect and correct. Furthermore, Shivakumar et al. [87] has reported that the SER for SRAM cells is expected to remain stable, while the SER for logic is steadily rising. Both these factors have motivated a flurry of research activities investigating solutions to protect the microproccessor core against transient faults. This body of related work will be addressed in Section 4.6.

Figure 4.1 shows the SER trend for a range of silicon technology generations reported in terms of *failures in time* (FIT[1]) per chip. Leveraging data presented by Shivakumar et al. [87], the SER trend for processor logic was scaled down to deep submicron technologies (similar to what is done by Borkar [18]) to generate the curve labeled *Nominal*. Note the exponential rise in SER with each new technology generation. Further exacerbating

[1]The number of failures observed per one billion hours of operation.

the SER challenge is the fact that in future technologies aggressive voltage scaling (both static and dynamic) will be required to meet power/thermal envelopes in the presence of unprecedented transistor densities. The curves, *Vscale_L*, *Vscale_M*, and *Vscale_H* illustrate the potential impact low, medium, and high amounts (respectively) of voltage scaling can have on SER.

Fortunately, a large fraction of transient faults are masked and do not corrupt actual program state. This masking can occur at the circuit, microarchitectural, or software levels. Our experiments, consistent with prior findings by Wang and Patel [109], show this masking rate to be around 92% collectively from all sources. Accounting for this masking, the raw SER at $45\,nm$ (the present technology node) translates to about one failure every month in a population of 100 chips. For a typical user of laptop/desktop computers this is likely imperceptible. However, in future nodes like $16\,nm$ the user-visible fault rate could be as high as one failure a day for every chip. The potential for this dramatic increase in the effective fault rate will necessitate incorporating soft error tolerance mechanisms into even low-cost, commodity systems.

4.2.2 Solution Landscape and Shoestring

As previously discussed, a soft error solution tailored for the commodity user space needs to be cheap, minimally invasive, and capable of providing *sufficient* fault coverage. Figure 4.2 is a conceptual plot of fault coverage versus performance overhead for the two types of fault detection schemes that form the foundation of Shoestring, one based on symptoms and the other on instruction duplication. The bottom region in this plot indicates the

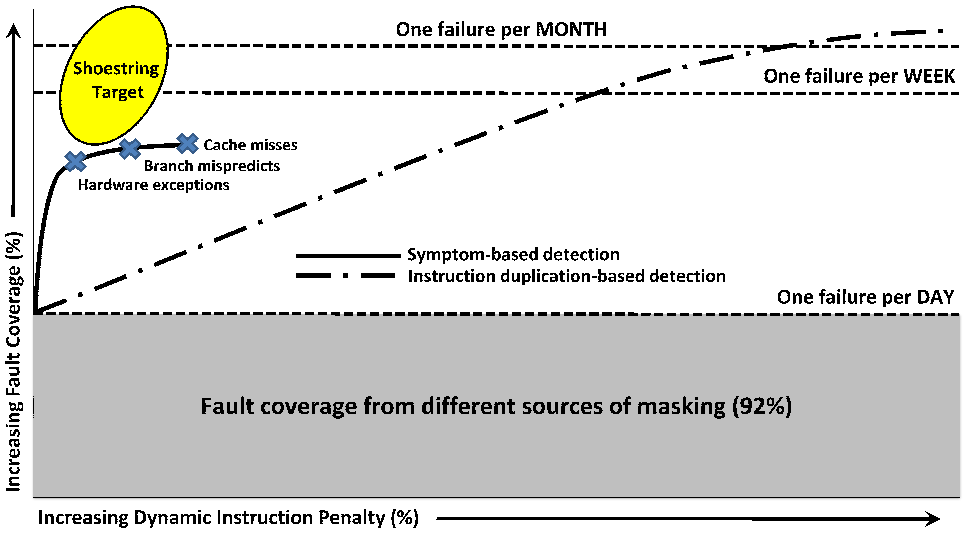


Figure 4.2: *Fault coverage versus dynamic instruction penalty trade-off for two existing fault detection schemes: symptom-based detection and instruction duplication-based detection. Also indicated is the region of the solution space targeted by Shoestring. The mapping of fault coverage to user-visible failure rate (dashed horizontal lines) is with respect to a single chip in a* $16\,nm$ *technology node with aggressive voltage scaling (Vscale_H).*

amount of fault coverage that results from intrinsic sources of soft error masking, available for free.

Of the remaining, unmasked faults, symptom-based detection is able to cover a significant fraction without incurring any appreciable overhead, mostly from detecting hardware exceptions. However, as a more inclusive set of symptoms are considered the overall coverage only improves incrementally while the performance overhead increases substantially. This is expected since these schemes relies on monitoring a set of rare events, treating their occurrence as symptomatic of a soft error, and initiating rollback to a lightweight checkpoint[2]. When the set of symptoms monitored is limited to events that rarely (if ever) occur under fault-free conditions (e.g., hardware exceptions) the performance overhead is negligible. However, when the set of symptoms is expanded to include more common events

[2]The checkpointing required for the symptom detection employed by Shoestring already exists in modern processors to support performance speculation (see Section 4.4).

like branch mispredicts and cache misses, the overhead associated with false-positives increases [111].

In contrast the coverage versus performance curve is far less steep for instruction duplication. Since instruction duplication schemes achieve fault coverage by replicating computation and validating the original and duplicate code sequences, the amount of coverage is easily tunable, with coverage increasing almost linearly with the amount of duplication.

The horizontal lines in Figure 4.2 highlight three fault coverage thresholds that map to effective failure rates of one failure per day, week, and month (in the context of a single chip in $16\,nm$ with aggressive voltage scaling *Vscale_H*). The fault coverage provided by the intrinsic sources of masking translates to about one failure a day, clearly unacceptable. To achieve a more tolerable failure rate of one fault per week or even month, comparable to other sources of failure in consumer electronics (e.g., software, power supply, etc.), the amount of fault coverage must be significantly improved. Note that although the symptom-based detection solution is both cheap and minimally invasive, it falls short of achieving these coverage thresholds. Similarly, although instruction duplication is capable of meeting these reliability targets, it does so by sacrificing considerable performance and power (from executing more dynamic instructions).

Although neither existing technique alone provides the desired performance and coverage tradeoffs, as a hybrid method, Shoestring is able to exploit the strengths of each, ultimately providing a technique that is optimally positioned within the solution space.

4.3 System Design

The main intuition behind Shoestring is the notion that near-perfect, "five-nines" reliability is not always necessary. In fact, in most commodity systems, the presence of such ultra-high resilience may go unnoticed. Shoestring exploits this reality by advocating the use of minimally invasive techniques that provide "just enough" resilience to transient faults. This is achieved by relying on symptom-based error detection to supply the bulk of the fault coverage at little to no cost. After this low-hanging fruit is harvested, judicious application of software-based instruction duplication is then leveraged to target the remaining faults that never manifest as symptoms.

To the first order, program execution consists of data computation and traversing the control flow graph (CFG). Correct program execution, strictly speaking, requires 1) that data be computed properly and 2) that execution proceeds down the right paths, i.e., compute the data *correctly* and compute the *correct* data. Working from this definition, previous software-based reliability schemes like SWIFT [75] have assumed that a program executes correctly (from the user's perspective) if all stores in the program are performed properly. This essentially redefines correct program execution as 1) storing the correct data (to the correct addresses) and 2) performing the right stores. Implicit is the assumption that the sphere of replication (SoR) [74], or the scope beyond which a technique cannot tolerate faults, is limited to the processing core. Faults in the caches and external memories are not addressed, but can be efficiently protected by techniques like ECC [44].

Shoestring, makes similar assumptions about SoR and correct program execution. However, unlike SWIFT [75] and other schemes, we are not targeting *complete* fault coverage.

```
// inpData is a global array
// process() is a global macro
1: index = 0
2: while (!stop)
3:   process(inpData[index])
4:   process(inpData[index + 1])
5:   process(inpData[index + 2])
6:   process(inpData[index + 3])
7:   index = index + 4
8:   stop = (index + 3) >= inpDataSize
9: end
10: // clean-up code
11: for (; index < inpDataSize; index++)
12:   process (inpData[index])
13: end
```

Figure 4.3: *A representative example of performance optimized code (loop unrolled).*

Relaxing the coverage constraint frees Shoestring from having to protect all portions of a program in order to guarantee correctness. This affords Shoestring the flexibility to only selectively protect those stores that are most likely to impact program output and least likely to already be covered by symptom detectors. Furthermore, we acknowledge that recent work by Wang et al. [110] has shown that as many as 40% of all dynamic branches are *outcome tolerant*. That is, they do not affect correct program behavior when forced down the incorrect path. The authors demonstrate that many of these so-called "Y-branches" are the result of partially dead control (i.e., they are data dependent and outcome tolerant the *majority* of the time). Leveraging this insight, Shoestring can also shed the overhead required to ensure that the CFG is *always* properly traversed. Instead, we focus on only protecting a subset of control flow decisions that impact "high-value" instructions.

Figure 4.3 shows a snippet of code where some manipulation of an array data structure is being performed. The computation is performed within a tight loop that uses the `process` macro to manipulate elements of the array `data`. Performance optimizations

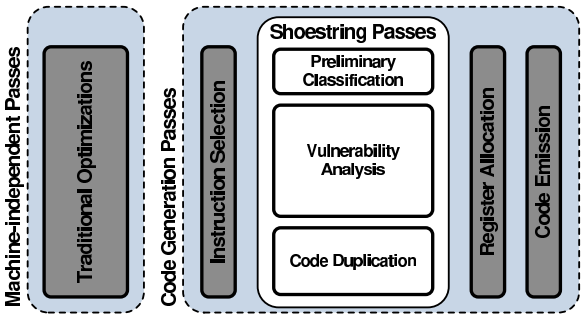


Figure 4.4: *A standard compiler flow augmented with Shoestring's reliability-aware code generation passes.*

cause the loop to be unrolled 4 times into lines 2 through 9. Additional cleanup code (lines 11 through 13) is also inserted to maintain program semantics. Note that in this example not all computation is essential for correct program behavior. The instruction at line 8 determines if the early loop termination condition is met. If the instruction(s) computing `stop` is (are) subjected to a transient fault, the unrolled loop could exit prematurely. Although this early exit degrades performance, program correctness is still maintained. In contrast, properly updating the variable `index` at line 7 is required for program correctness (assuming of course that `inpData` is a user-visible variable). However, since `index` is also used as a base address to access `inpData`, there is a significant probability that a fault corrupting `index` would manifest as a symptomatic memory access exception. Given the proper symptom-based detection scheme, this could decrease the *effective* vulnerability of the computation at line 7. Identifying instructions critical to program correctness and pruning from this set those instructions that are already "covered" by symptom-based detection is the focus of the remainder of this section.

4.3.1 Compiler Overview

Implementing the most cost effective means of deploying instruction duplication requires detailed compiler analysis. Shoestring introduces additional reliability-aware code generation passes into the standard compiler backend. Figure 4.4 highlights these passes in the context of typical program compilation. Shoestring's compilation passes are scheduled after the program has already been lowered to the machine-specific representation but before register allocation.

The first two passes, *Preliminary Classification* and *Vulnerability Analysis*, are designed to categorize instructions based on their expected behavior in the presence of a transient fault. These categories are briefly described below.

- **Symptom-generating**: these instructions, if they consume a corrupted input, are likely to produce detectable symptoms.
- **High-value**: these instructions, if they consume a corrupted input, are likely to produce outputs that result in user-visible program corruption.
- **Safe**: these instructions are naturally covered by symptom-generating consumers. For any safe instruction, I_S, the expectation is that if a transient fault is propagated by I_S, or arises during its execution, there is a high probability that one of its consumers will generate a symptom within an acceptable latency S_{lat}.
- **Vulnerable**: all instructions that are not safe are considered vulnerable.

Following the initial characterization passes, a third pass, *Code Duplication*, performs selective, software-based instruction duplication to protect instructions that are not inher-

ently covered by symptoms. This duplication pass further minimizes wasted effort by protecting only the high-value instructions, those likely to impact program output. By only duplicating instructions that are along the dataflow graph (DFG) between safe and high-value instructions, the performance overhead can be dramatically reduced without significantly impacting reliability.

The following sections describe the details of the heuristics used in the analysis and duplication passes.

4.3.2 Preliminary Classification

Shoestring's initial characterization pass iterates over all instructions in the program and identifies symptom-generating and high-value instructions. For clarity, this classification is described as a separate compiler pass. However, in practice the identification of symptom-generating and high-value instructions can be performed as part of the vulnerability analysis pass.

4.3.2.1 Symptom-generating Instructions

The symptom events considered by prior symptom-based detection work can be broadly separated into the following categories [111, 48]:

- **ISA-defined Exceptions**: these are exceptions defined by the instruction set architecture (ISA) and must already be detected by any hardware implementing the ISA (e.g., page fault or overflow).

- **Fatal Exceptions**: these are the subset of the ISA-defined exceptions that never occur under normal user program execution (e.g., segment fault or illegal opcode).

- **Anomalous Behavior**: these events occur during normal program execution but can also be symptomatic of a fault (e.g., branch mispredict or cache miss).

The relative usefulness of symptoms in each of these categories is dependent on how strongly their appearance is correlated with an actual fault. Ideal candidates occur very rarely during normal execution, minimizing overhead due to false positives, but always manifest in the wake of a fault. Therefore, to maximize the overhead-to-coverage tradeoff the experiments in Section 4.5 evaluate a Shoestring implementation that only considers instructions that can elicit the second category of fatal, ISA-defined exceptions as potentially symptom generating. Since these are events that during the normal execution of user programs never arise, they incur no performance overhead in the absence of faults.

Although additional coverage can be gleaned by evaluating a more inclusive set of symptoms, prior work has shown that the additional coverage often does not justify the accompanying costs. For example, Wang and Patel [111] presented results where using branch mispredictions on high-confidence branches as a symptom gained an additional 0.3% of coverage with an 8% performance penalty. Other non-fatal symptoms like data cache misses also have similar coverage and overhead profiles.

4.3.2.2 High-value Instructions

Ideally, we would like to only consider instructions that impact program output as high-value. However, given that the analysis necessary to provably make such determinations is

impractical, if not intractable, heuristics must be employed. Currently, any instructions that can potentially impact global memory is considered high-value. In addition, any instructions that can produce arguments passed to function calls (especially library calls) are also included. To provide a truly transparent solution, Shoestring, at present, assumes that no user annotations are available to assist in instruction classification. Future extensions could leverage techniques from information-flow theory [106, 54] to further refine the instruction selection process or even exploit the work by Li and Yeung [50] to prune instructions that only impact "soft" program outputs. Although investigating more sophisticated heuristics for identifying high-value instructions is a very promising avenue of future work, it was not attempted in this thesis.

4.3.3 Vulnerability Analysis

After the preliminary instruction classification is complete, Shoestring analyzes the vulnerability of each instruction to determine whether it is safe. As stated previously, a safe instruction, I_S, is one with enough symptom-generating consumers such that a fault corrupting the result of I_S is likely to exercise a symptom within a fixed latency S_{lat}. For each instruction, the number of symptom-generating consumers is first tabulated based on distance. For a given producer (I_p) and consumer (I_c) pair, we define the distance, $D_{p,c}$, as the number of intervening instructions between I_p and I_c within the statically scheduled code. It is used as a compile-time estimate of the symptom detection latency if the consumer, I_c, were to trigger a symptom. For a given instruction, I, if the number of symptom-generating consumers at distance i is N_i, then I is considered safe if $N_{tot} = \sum_{i=1}^{S_{lat}} N_i$ is greater than a fixed threshold S_t. The value for the threshold parameter S_t controls the selectivity of safe

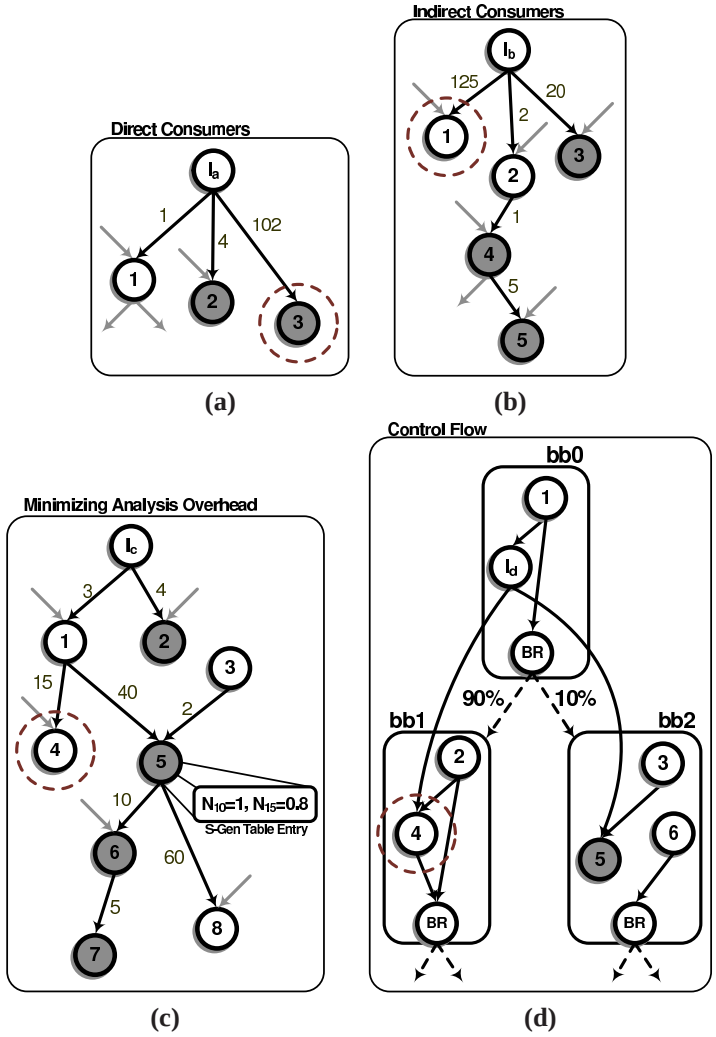


Figure 4.5: *Example data flow graphs illustrating Shoestring's vulnerability analysis. The data flow edge numbers represent the distance between two instructions in the statically scheduled code. Shaded nodes represent symptom-generating instructions and dashed circles highlight high-value instructions. Dashed edges in (d) represent control flow.*

instruction classification and can be used to trade off coverage for performance overhead (see Section 4.5).

Figure 4.5 and the corresponding case studies illustrate how the vulnerability analysis heuristic is applied for a few sample DFGs. The numbers along the data-flow edges represent the distance, $D_{p,c}$, between the two nodes (instructions). Shaded nodes indicate symptom-generating instructions, and nodes highlighted by a dashed circle are high-value instructions. For all the case studies, $S_{lat} = 100$ and $S_t = 2$.

4.3.3.1 Case Study 1: Direct Consumers

In Figure 4.5a, instruction I_a is being analyzed for safe-ness. Instructions 1, 2, and 3 are all direct consumers of I_a. Instructions 2 and 3 have already been identified as symptom-generating instructions and 3 is also a high-value instruction. In this example, I_a would be classified as vulnerable because it only has one symptom-generating consumer within a distance of 100 (S_{lat}), instruction 2.

4.3.3.2 Case Study 2: Indirect Consumers

Figure 4.5b presents a more interesting example that includes direct as well as indirect consumers as we analyze I_b. As with direct consumers, indirect consumers that have been identified as symptom-generating also contribute to the N_{tot} of I_b. However, their contribution is reduced by a scaling factor S_{iscale} to to account for the potential for *partial* fault masking.

In Figure 4.5b, instructions 3, 4, and 5 are all symptom generating consumers of I_b. Since 3 is a direct consumer, any fault that corrupts the result of I_b will cause instruction 3 to generate a symptom (probabilistically of course). However, the same fault would have to propagate through instruction 2 before it reaches the indirect consumer, instruction 4. This allows for the possibility that the fault may be masked by 2 before it actually reaches 4. For example, if the soft error flipped an upper bit in the result of I_b and instruction 2 was an `AND` that masked the upper bits, the fault would never be visible to instruction 4, reducing its ability to manifest a symptom. However, instruction 1 would still consume the tainted value and potentially write it out to memory, corrupting system state. Therefore, due to the

potential for masking, an indirect consumer is less likely than a direct consumer to cover the exact same fault. Ultimately with respect to I_b in Figure 4.5b, given an $S_{iscale} = 0.8$, we have $N_{20} = 1$, $N_3 = 0.8$, and $N_5 = 0.64$. Since $N_{tot} = \sum_{i=1}^{100} N_i = 2.44$ and is greater than the threshold S_t of 2, I_b is classified as safe.

4.3.3.3 Case Study 3: Minimizing Analysis Overhead

Figure 4.5c presents a more complete example and further illustrates how memoization is used to avoid redundant computation. Rather than identifying indirect symptom-generating consumers recursively for every instruction, we maintain a global *symptom-generation table* of N_i values for every instruction. By traversing the DFG in a depth-first fashion, we guarantee that all the consumers of an instruction are processed before the instruction itself is encountered. Creating an entry in the symptom-generation table (labeled S-Gen Table in the Figure 4.5c) for every instruction as it is being analyzed ensures that each node in the DFG only needs to be visited once[3].

For example, assuming the vulnerability analysis begins with I_c, Shoestring analyzes the instructions in the following order, 4, 7, 6, 8, 5, 1, 2 and eventually marks I_c as safe. When the analysis pass reaches instruction 3 it can determine its classification directly, without identifying any of its indirect consumers, since the symptom-generation table entry for instruction 5 was already populated during the analysis pass for I_c. The corresponding table entry for 3 is computed by scaling all N_i entries for 5 by S_{iscale}, adjusting the corresponding distances by adding 2, and finally accounting for the symptom-generating potential of instruction 5 itself. The table entry for instruction 3 would then

[3] Although this optimization is beneficial for programs with large functions, even a naive recursive analysis for the SPEC2K applications evaluated in this work did not appreciably increase compilation time.

contain $N_2 = 1, N_{12} = 0.8, N_{17} = 0.64$ and instruction 3 would subsequently also be classified as safe.

Obviously, this depth-first traversal is complicated in the presence of loops (not present in Figure 4.5c) where circular dependencies can exist and the traversal could loop indefinitely never reaching a leaf node. Consequently, whenever Shoestring encounters a loop it forces the depth-first traversal to backtrack when the distance between the instruction currently being processed and the instruction at the bottom of the loop exceeds S_{lat}. This guarantees all relevant symptom-generating consumers are accounted for while also ensuring forward progress.

4.3.3.4 Case Study 4: Control Flow

The examples examined so far have been limited to analyzing instruction DFGs and control has to a large extent been ignored. Although Shoestring takes a relaxed approach with respect to *enforcing* correct control flow, branching is taken into consideration when performing vulnerability analysis. Figure 4.5d shows an example where the instruction being analyzed, I_d, is in a basic block (bb0) that has a highly biased branch. In this scenario, although instruction 5 is a symptom-generating consumer, because it is in a basic block (bb2) that is unlikely to be executed, it will not provide dependable coverage for I_d. Therefore, the contribution of every consumer to N_i is scaled by their respective execution probabilities. These execution probabilities are extracted from profiled execution (provided to the experiments in Section 4.5), or when profile data is unavailable, generated from static approximations.

Lastly, although Wang et al. [110] showed that execution down the wrong direction of many branches ultimately reconverges with the correct execution path, in Figure 4.5d if the branch terminating bb0 is corrupted causing execution to proceed to bb2 instead of bb1, there is no time for control to reconverge before instruction 4 potentially corrupts system state. Therefore, Shoestring also selectively protects (by duplicating input operand chains) all branches that have a control-dependence edge with a high-value instruction. For sake of brevity, the standard algorithm for identifying control-dependence edges will not be presented here but it is important to note that not all branches that can influence whether instruction 4 is executed will be protected. Only those branches that are effectively the "nearest" to instruction 4 will possess the requisite control-dependence edges and be protected, leaving the rest (which are further away and more likely to reconverge) vulnerable.

4.3.4 Code Duplication

The process of inserting redundant code into a single thread of execution has been well studied in the past [76, 64]. In general, this process involves duplicating all computation instructions along the path of replication and inserting comparison instructions at synchronization points (e.g., at memory and control flow instructions) to determine if faults have manifested since the last comparison was performed. This section will highlight how Shoestring's code duplication pass departs from this existing practice. The reader is encouraged to examine prior work for a detailed description of the mechanics of code duplication.

The code duplication pass begins by selecting a single high-value instruction, I_{HV}, from the set of all high-value instructions. It then proceeds to recursively duplicate all

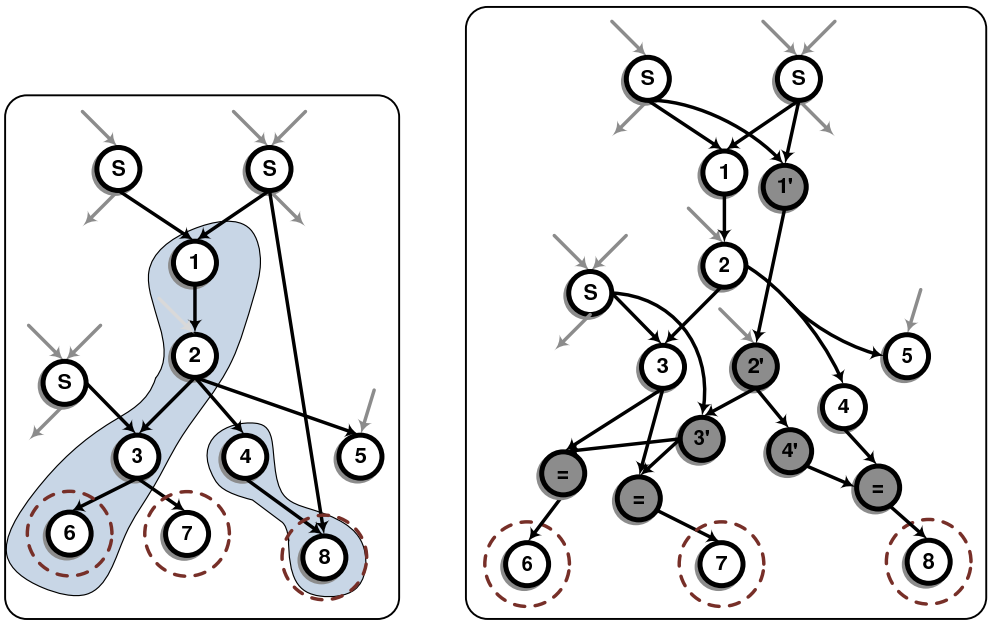


Figure 4.6: *Example data flow graph illustrating Shoestring's code duplication pass. Nodes labeled with an "S" represent safe instructions and dashed circles highlight high-value instructions. In (a) the shaded portions of the graph represent code duplication chains. (b) shows the new DFG with all duplicated instructions inserted as shaded nodes. Nodes labeled with an "=" represent checker instructions.*

instructions that produce values for I_{HV}. This duplication is terminated when 1) no more producers exist, 2) a safe instruction is encountered, or 3) the producer has already been previously duplicated. In all cases, it is guaranteed that every vulnerable instructions that could possibly influence data consumed by I_{HV} is duplicated. Comparison instructions are inserted right before I_{HV} to verify the computation of each of its input operands.

Figure 4.6a presents a section of a DFG with three high-value instructions (nodes 6, 7, and 8), three safe instructions (nodes labeled with an "S"), and five vulnerable instructions (nodes 1-5). For this example, we start with instruction 6 and begin by duplicating its producer, instruction 3. Next, we attempt to duplicate the producers for 3 and notice that one of the producers has been classified as safe and terminate code duplication on that path. The other producer for 3 (instruction 2), however, is vulnerable so we duplicate 2 and continue

along its producer chain duplicating instruction 1 as well. Subsequent attempts to duplicate 1's consumers encounters safe instructions, at which point all vulnerable code relevant to high-value instruction 6 has been duplicated. Shoestring then moves on to the next high-value instruction and repeats the process with instruction 7. At this point, instruction 3 has already been duplicated as a result of protecting instruction 6 so nothing needs to be done. Next, instruction 8 is considered, resulting in the duplication of instruction 4.

Figure 4.6b shows the new DFG with all the duplicated instructions (shaded nodes) and checkers ("=" nodes) inserted. Note that both high-value instructions 6 and 7 each have their own checker to compare the results from instruction 3 and its redundant copy 3'. Although both 6 and 7 consume the same value, only relying on a single checker at instruction 6 to detect faults that corrupt 3's result could leave 7 vulnerable to faults that corrupt the result of 3 after 6 has already executed. Depending on how far apart 6 and 7 execute, this vulnerability window could be significant. Nevertheless, in situations where high-value instructions with common producers also execute in close proximity, the need for duplicate checkers can also be avoided. However, this optimization is not investigated in this work.

4.4 Experimental Methodology

Given that this chapter is targeting coverage of faults induced by soft errors on *commodity* processors, we would ideally conduct electron beam experiments using real hardware running code instrumented by Shoestring. Given limited resources a popular alternative to beam experiments is statistical fault injection (SFI) into a detailed register transfer lan-

guage (RTL) processor model. However, since Shoestring exploits fault masking at the application level, full program simulation is also required. Since simulating realistic benchmarks on RTL models is extremely slow, a common practice in the literature is to rely on microarchitectural-level simulators to provide the appropriate compromise between simulation fidelity and speed.

4.4.1 Fault Model and Injection Framework

The fault injection results presented in this chapter are generated using the PTLsim x86 microarchitectural simulator [120]. PTLsim is able to run x86 binaries on the native (host) machine as well as within a detailed microarchitectural simulator. Being able to effectively switch between native hardware execution and microarchitectural simulation on-the-fly enables fast, highly detailed simulations. We simulated a modern, high performance, out-of-order processor modeled after the AMD K8 running x86 binaries. The details of the processor configuration can be found in Table 4.1.

The fault model we assume is a single bit flip within the physical register file. Although they are not explicitly modeled, most faults in other portions of the processor eventually manifest as corrupted state in the register file, making it an attractive target for injection studies[4]. Furthermore, Wang et al. [112] showed that the bulk of transient-induced failures are dominated by corruptions introduced from injections into the register file. Nevertheless, our methodology may not fully capture the ability of Shoestring to handle faults from combinational logic with large fanouts.

[4]Only performing fault injections into the register file is a limitation of our evaluation infrastructure, not a limitation of Shoestring's fault coverage abilities. In reality Shoestring will detect soft errors that strike other parts of the processing core as well.

Table 4.1: *Processor details (configured to model an AMD-K8).*

Processor core @ 2.2GHz	
Fetch queue size	36 entries
Reorder buffer size	72 entries
Issue queue size	16 entries
Issue width	16 entries
Fetch/Dispatch/Writeback/ Commit width	3
Load/Store queue size	44 entries (each)
Physical register file size	128 entries
Physical register file size	128 entries
Memory	
L1-I/L1-D cache	64KB, 2-way, 3 cycle lat
L2 cache (unified)	1MB, 16-way, 10 cycle latency
DTLB/ITLB	32 entries (each)
Main memory	112 cycle lat

The experimental results shown in this chapter are produced with Monte Carlo simulations. At the start of each Monte Carlo trial a random physical register bit is selected for injection. It has been shown that the memory footprint of SPEC2K applications are significantly smaller than the full size of a 64-bit virtual address space. Allowing faults to occur in any of the 64-bits of a register would increase the likelihood of it resulting in a symptomatic exception [111], and consequently being covered by Shoestring. Therefore, although PTLsim simulates a 64-bit register file, we limit our fault injections to only the lower 32 bits to avoid artificially inflating Shoestring's coverage results.

Once an injection site is determined, program simulation is allowed to run in native mode (running on real hardware) until it reaches a representative code segment (identified using SimPoint [84] and manual source code inspection). At this point PTLsim switches to detailed mode and warms up the microarchitectural simulator. After a randomly selected number of cycles has elapsed, a fault is induced at the predetermined injection site. De-

tailed simulation continues until 10M instructions commit, at which time PTLSim copies architectural state back to the host machine and resumes simulating the remainder of the program in native mode. This of course assumes that the fault did not result in a fatal exception or program crash prior to 10M instructions. At the end of every simulation the log files are analyzed to determine the outcome of the Monte Carlo run as described in the next section.

4.4.2 Outcome Classification

The result of each Monte Carlo trial is classified into one of four categories:

1. **Masked**: the injected fault was naturally masked by the system stack. This includes trials where the fault was architecturally masked as well as those that were masked at the application level.

2. **Covered by symptoms**: the injected fault resulted in anomalous program behavior that is symptomatic of a transient fault. For these trials it is assumed that system firmware is able to trigger recovery from a lightweight checkpoint. The details of this assumed checkpointing mechanism are described in the next section.

3. **Covered by duplication**: faults in this category were the result of injecting code that was selectively duplicated by Shoestring. During these trials the comparison instructions at the end of every duplication chain would trigger a function call to initiate recovery.

4. **Failed**: In this work the definition of failure is limited to only those simulation runs which completed (or prematurely terminated) with user-visible data corruptions.

Although the definition for failure used in this chapter may seem unconventional, it is consistent with recent symptom-based work and is the most appropriate in the context of evaluating Shoestring. The main premise behind the Shoestring philosophy is that the cost of ensuring reliable computation can be reduced by focusing on covering only the faults that are ultimately noticeable by the end user. Therefore, the figure of merit is not the number of faults that propagated into microarchitectural (or architectural) state, but rather the fraction that actually resulted in user-visible failures.

4.4.3 System Support

As briefly discussed in the previous section, Shoestring (and symptom-based schemes in general) relies on the ability to rollback processor state to a clean checkpoint. The results presented in Section 4.5 assume that in modern/future processors a mechanism for recovering to a checkpointed state of 10-100 instructions in the past will already be required for aggressive performance speculation. Consistent with Wang and Patel [111], Shoestring assumes that any fault that manifests as a symptom within a window of 100 committed instructions (micro-ops, not x86 instructions) can be safely detected and recovered. The proper selection of the S_{lat} parameter described in Section 4.3.3 is closely tied to the size of this checkpointing window. Only those consumers that can be expected to generate a symptom within this window are considered when identifying safe instructions. Similarly, faults that are detected by instruction duplication would also trigger rollback and recovery.

The results presented in Section 4.5 assume a checkpointing interval, and consequently an S_{lat} value, of 100. Although this small window may seem modest in comparison to checkpointing intervals assumed by other work, most notably Li et al. [48], it is the most ap-

propriate given Shoestring's goals of providing minimally invasive, low cost protection. Increasing the size of this window would unfairly inflate the coverage provided by Shoestring since accommodating large checkpointing intervals requires substantial hardware/software overhead. However, if large checkpointing intervals eventually find their way into mainstream processors, the heuristics used by Shoestring can be easily tuned to exploit this additional support and provide even greater coverage.

The compilation component of Shoestring is implemented in the LLVM compiler [47]. The reliability-aware code generation passes described in Section 4.3.1 are integrated as part of the code generation backend. Six applications from the SPEC2K integer benchmark suite (*gzip*, *mcf*, *crafty*, *bzip2*, *gap*, and *vortex*) are used as representative workloads in our experiments and are compiled with standard -O3 optimizations. To minimize initial engineering effort, we only evaluated benchmarks from the SPEC2K suite that both 1) compiled on standard LLVM (without modifications for Shoestring) and 2) simulated correctly on PTLSim "out-of-the-box". They were not handpicked because they exhibited desirable behavior. Similarly, to minimize engineering effort we do not apply Shoestring to library calls. The common practice in the literature is to assume that dynamically linked library calls are protected by some other means, i.e., outside the SoR (see Section 4.3) [75]. The results presented in Section 4.5 adheres to the same practice and avoids injections into library calls.

Lastly, due to limitations of our evaluation framework we do not study Shoestring in the context of multithreaded/multicore environments. Given that we treat the cache as outside our SoR the majority of challenges posed by the shared memory in multithreaded/multicore systems would not impact the efficacy of Shoestring. However, the larger memory foot-

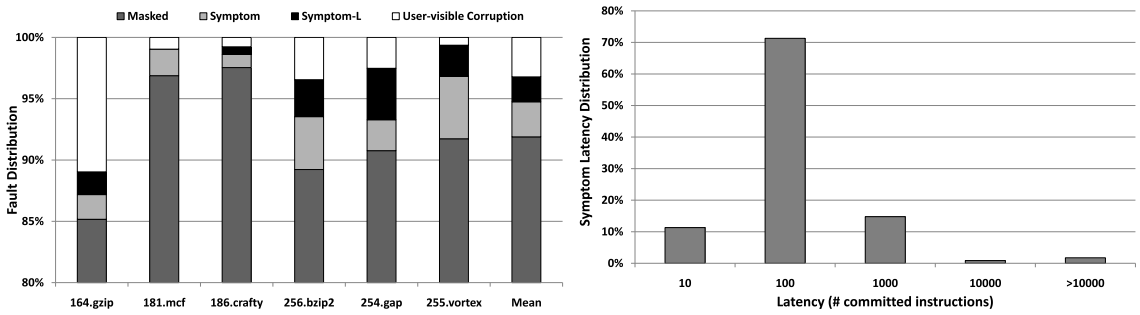
prints of multithreaded applications could potentially attenuate the coverage due to a reduction in the performance of memory access symptoms. The greater resource/register utilization in simultaneous multithreaded systems could also reduce the amount of masking we see from faults that strike dead/free registers. Lastly, identifying the exact core which triggered the symptom, as well as orchestrating the checkpoint rollback and recovery, is more challenging when multiple threads running on different cores are interacting and sharing data. However, these challenges are beyond the scope of the current thesis and are left as interesting directions for future work.

4.5 Evaluation and Analysis

This section begins with results from an initial fault injection campaign to quantify the amount of opportunity available for Shoestring to exploit. We then proceed to examine the compilation heuristics described in Section 4.3. Finally, we present and analyze the fault coverage and runtime overheads for Shoestring. All experimental results included in this section are derived from >10k Monte Carlo trials.

4.5.1 Preliminary Fault Injection

Figure 4.7 presents the results of initial fault injection trials. The purpose of this preliminary experiment was to identify the amount of faults that are inherently masked throughout the entire system stack. The accumulation of all these sources of masking, from the microarchitecture up through the application layer, is essentially the amount of "coverage" that is available for free. This is shown as the *Masked* segment on the stacked bars and



(a) Symptom-based fault coverage **(b)** Latency distribution of symptom-based detection.

Figure 4.7: *Results of preliminary fault injection experiments. (a) shows the percentage of faults that are intrinsically masked (*Masked*), covered by symptoms (*Symptom*), covered by long-latency symptoms (*Symptom-L *), or result in user-visible failures (*User-visible Corruption*).*

corresponds to roughly 91.9% on average across the benchmarks. Symptoms account for another 4.9% and actual user-visible failures account for the remaining 3.2%.

As mentioned in Section 4.3 symptom-based coverage is only useful if symptoms are triggered within a small window of cycles, S_{lat}, following a fault. If the symptom latency exceeds S_{lat} then the likelihood that state corruption can occur before a symptom is manifested increases. Note now the portions of the chart labeled as *Symptom-L*. These segments are the fraction of trials that lead to symptoms but did so only after the 100 instruction S_{lat} window expired. Without more expensive check-pointing to accommodate longer latencies, the Symptom-L cases must also be considered failures. Figure 4.7b examines these symptom-generating trials from a different perspective, as a distribution based on detection latency. Although the majority of symptoms do manifest within the 100 instruction threshold, roughly 14.7% would require a much more aggressive checkpointing scheme (1000 instructions) than what is needed for performance speculation alone. Furthermore, the remaining 2.6%, with latencies of more than 10,000 instructions could not be exploited without being accompanied by heavyweight, software-based checkpointing (and its attendant

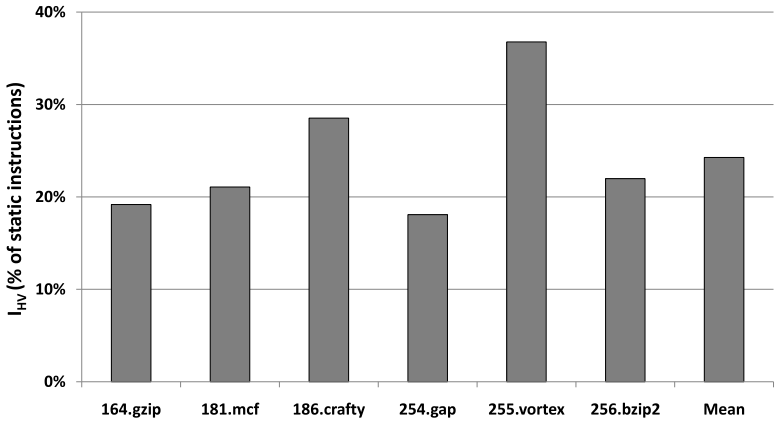


Figure 4.8: *Percentage of static instructions classified as high-value (I_{HV}).*

costs). The remainder of the chapter assesses Shoestring's ability to minimize user-visible corruptions by integrating symptom-based coverage with intelligent software-based code duplication.

4.5.2 Program Analysis

4.5.2.1 High-value Instructions

To gain insight into how selective instruction duplication is actually applied by Shoestring, we examine the heuristics described in Section 4.3 in the context of our SPEC2K workloads. Figure 4.8 shows the percentage of instructions identified as high-value within each benchmark. As discussed in Section 4.3.2.2, only instructions that can potentially modify global memory or produce values for function calls are considered high-value. On average roughly 24.3% of all static instructions meet this criteria and become the focus of Shoestring's code duplication efforts.

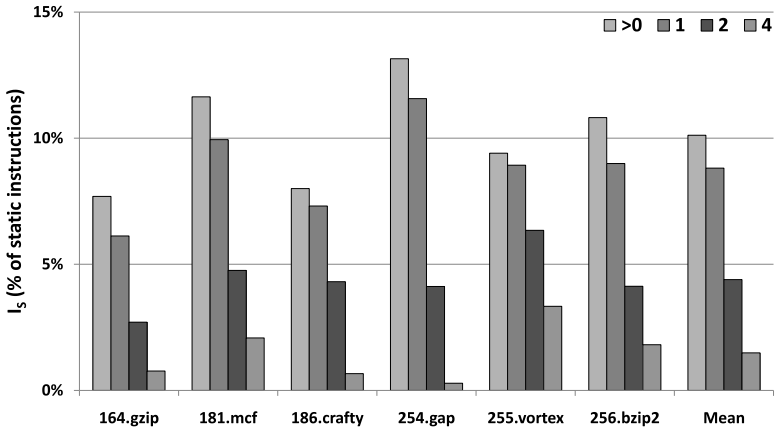


Figure 4.9: *Percentage of static instructions classified as safe as S_t is varied (I_S).*

4.5.2.2 Safe Instructions

Figure 4.9 presents the percentage of instructions classified as safe, as a function of the heuristic parameter S_t (Section 4.3.3). A value of n for S_t indicates that for an instruction to be considered safe (i.e., covered by symptom-generating consumers) it must possess at least n consumers within a distance of S_{lat} instructions along any potential path of execution. Note that on average, even with S_t relaxed to allow for any non-zero threshold ($>$0) only 10.1% of static instructions are classified as safe. This is mainly due to our conservative decision to only consider potential ISA-excepting instructions as candidates for symptom-generating consumers. A more aggressive heuristic could potentially identify more safe instructions if the set of symptoms that it monitored was more inclusive. However, this would come at the cost of performance-degrading false positives.

For this, and all subsequent, experiments the value of S_{lat} was fixed at 100 instructions as explained in Section 4.4.3. A value of 0.9 for S_{iscale} (Section 4.3.3.2) was also empirically determined to produce the best heuristic behavior, and is fixed for all experiments.

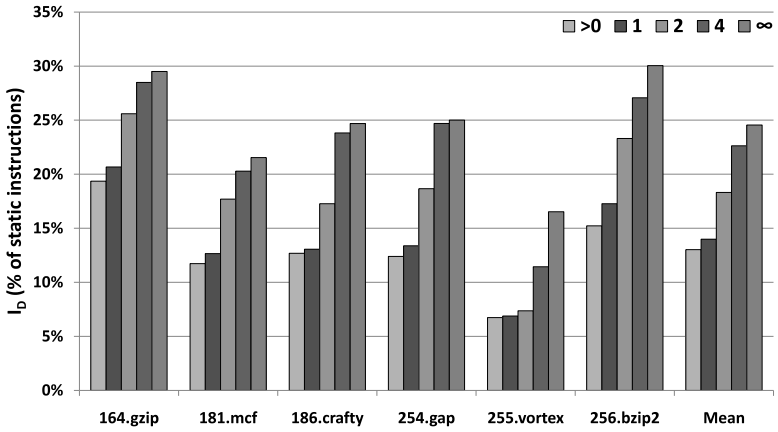


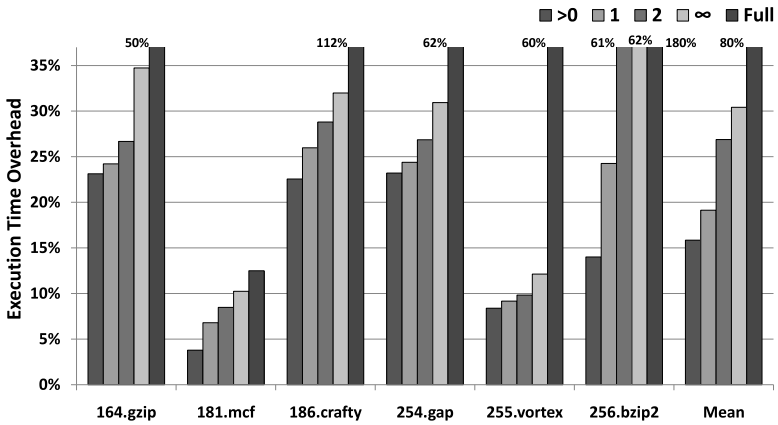
Figure 4.10: *Percentage of static code duplication performed by Shoestring as S_t is varied (I_D).*

4.5.2.3 Duplicated Instructions

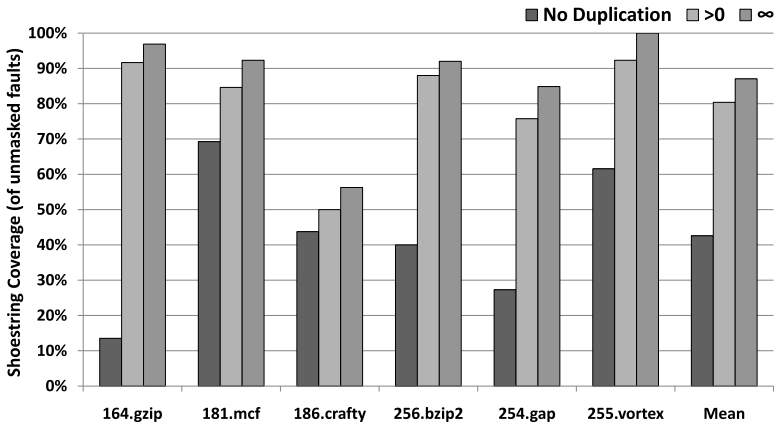
Figure 4.10 shows the percentage of (static) instructions that are duplicated by Shoestring as S_t is swept from >0 to ∞. Note the direct relationship between S_t and the number of duplicated instructions. This is attributable to the fact that code duplication begins at high-value instructions and terminates at the first safe instruction encountered (see Section 4.3.4). Therefore, the fewer instructions that are classified as safe, the less likely a duplication chain will terminate early. In the extreme case when $S_t = \infty$ no instructions are classified as safe. This results in fully duplicating producer chains for every high-value instruction.

4.5.3 Overheads and Fault Coverage

Next we examine the runtime overhead of the binaries that have been protected by Shoestring's selective code duplication. Figure 4.11a shows that as S_t is varied from >0 to ∞ the performance overhead of Shoestring ranges from 15.8% to 30.4% (obtained from native simulation on an Intel Core 2 processor). The execution overheads for a full soft-



(a) Runtime performance overhead.



(b) Fault coverage.

Figure 4.11: *Fault coverage and runtime performance overheads for Shoestring as a function of* S_t.

ware duplication scheme, SWIFT [75], are also included for the sake of comparison[5](bars labeled *Full*). Since Shoestring is positioned as a reliability "on the cheap" solution, maintaining low runtime overhead is of paramount importance. To evaluate the amount of coverage that can be obtained by Shoestring with the least amount of performance overhead, for the remainder of this section the value of S_t is fixed at >0.

Figure 4.11b presents the coverage results for this Shoestring configuration. Also included are coverage numbers for no instruction duplication, *No Duplication*, and $S_t = \infty$

[5]The overheads we cite from SWIFT [75] are conservative considering they targeted a wide VLIW machine and would incur substantially more overhead given a less overprovisioned processor.

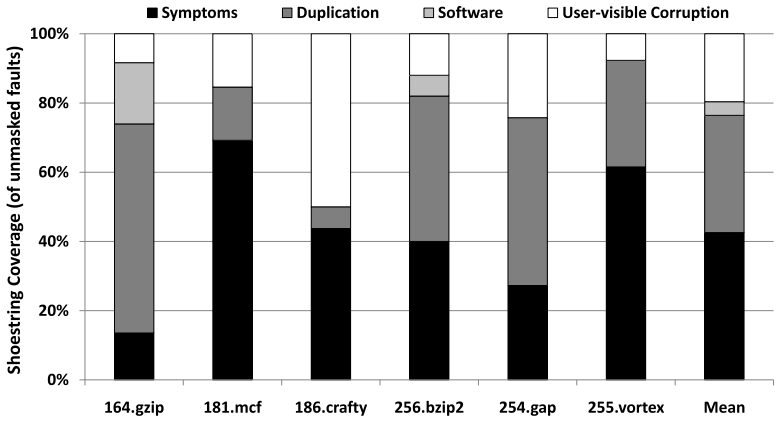


Figure 4.12: *Detailed coverage breakdown for Shoestring configured with S_t fixed at >0.*

to illustrate where the proposed solution sits with respect to the upper and lower bounds. Coverage numbers for other values of S_t were not included because of the requisite simulation effort. Although investigating more sophisticated heuristics for instruction classification and vulnerability analysis has the potential to garner even more coverage, note that Shoestring is already able to recover from 80.4% of the failures that would have otherwise gone unmasked and caused user-visible data corruptions.

Lastly, Figure 4.12 takes a closer look at the fault coverage achieved by Shoestring. The stacked bars highlight the individual components contributing to Shoestring's total fault coverage. Note that on average, selective duplication covers an additional 33.9% of the unmasked faults that would have slipped by a symptom-only based scheme. Notice also the segment labeled *Software*. This category, only significant for *gzip* and *bzip2*, is the result of assertions placed in the source code that actually detect the erroneous behavior of the program following a fault injection. This observation suggests that perhaps only a modest investment is required to enhance conventional assertion checks with information that could improve Shoestring's program analysis.

Table 4.2: *Shoestring compared to existing solutions for soft errors. "HW":hardware, "SW": software, "nMR": n-modular redundancy, "RMT": redundant multithreading, "RF-only": register file protection only.*

Solution	HW Support	SW Support	Performance Overhead	Area Overhead	Coverage
nMR	YES	NO	LOW	VERY HIGH	VERY HIGH
RMT	YES	MAYBE	HIGH	HIGH	VERY HIGH
SW instruction duplication	NO	YES	HIGH	NONE	HIGH
Symptom-based detection	NO	NO	LOW	NONE	MODERATE
RF-only	YES	NO	LOW	MODERATE	MODERATE
Shoestring	NO	YES	LOW	NONE	HIGH

4.6 Related Work

This section examines Shoestring in the context of previous work. Table 4.2 presents a quick overview of where Shoestring sits within the soft error solution space. The ability to achieve high levels of fault coverage with very low performance overhead, all without any specialized hardware, sets it apart from previously proposed schemes. Each category of alternative solutions is addressed in detail below.

n-Modular Redundancy (nMR): Spatial or temporal redundant execution has long been a cornerstone for detecting soft errors, with hardware DMR (dual-modular redundancy) and TMR (triple-modular redundancy) being the solutions of choice for mission-critical systems. However, the cost of such techniques has relegated them to the high-budget server and mainframe domains (e.g., HP NonStop series [14] and IBM zSeries [11] machines). DIVA [8] is a less expensive alternative to full hardware duplication, utilizing a small checker core to monitor the computations performed by a larger microprocessor.

Rather than employing full hardware replication, recent work has also been interested in using smaller, lightweight hardware structures to target individual components of the processor. Argus [55] relies on a series of hardware checker units to perform online invariant checking to ensure correct application execution. In [72] Reddy and Rotenberg propose simple checkers that verify the functionality of decode and fetch units by comparing dynamically generated signatures, for small traces of identical instructions. They extend this idea in [73] by introducing additional checkers for other microarchitectural structures.

Although the area overhead of solutions like DIVA and Argus are significantly lower than full DMR, they still remain an expensive choice for commodity systems. Nevertheless, these nMR (and partial nMR) solutions provide greater fault coverage than Shoestring and can provide bounds on detection latency.

Redundant Multithreading (RMT): The introduction of simultaneous multithreading (SMT) capabilities in modern processors gave researchers another tool for redundant execution. Rotenberg's paper on AR-SMT [80] was the first to introduce the concept of RMT on SMT cores. The basic idea was to use the processor's extra SMT contexts to run two copies of the same thread, a leading thread and a trailing thread. The leading thread places its results in a buffer, and the trailing thread verifies these results and commits the executed instructions. Subsequent works improved upon this scheme by optimizing the amount of duplicated computation introduced by the redundant thread [74, 37, 71]. RMT has also been attempted at the software level by Cheng et al. [108]. This eliminates the need for architectural modifications to support RMT, and relies on the compiler to generate redundant threads that can run on general-purpose chip multiprocessors. With the advent of multicore, RMT solutions have evolved into using two or more discrete cores within a CMP to mimic

nMR behavior. Reunion [91] and Mixed-mode reliability [115] are two recent proposals that allow idle cores within a CMP to be leveraged for redundant thread execution. The chief attraction of RMT approaches is the high coverage they can provide. The drawbacks of RMT include significant throughput degradation (loss of an SMT context or an entire core), hardware complexity/overhead, and potentially double the power consumption of non-RMT execution.

Software instruction duplication: Redundant execution can also be achieved in software without creating independent threads as shown by Reis et al. [75]. They proposed SWIFT, a fully compiler based software approach for fault tolerance. SWIFT exploits wide, underutilized processors by scheduling both original and duplicated instructions in the same execution thread. Validation code sequences are also inserted by the compiler to compare the results between the original instructions and their corresponding duplicates. CRAFT [76] and PROFIT [76] improve upon the SWIFT solution by leveraging additional hardware structures and architectural vulnerability factor (AVF) analysis [60], respectively. As in the case of RMT, compiler-based instruction duplication also delivers nearly complete fault coverage, with the added benefit of requiring little to no hardware cost. However, in order to achieve this degree of fault coverage, solutions like SWIFT can more than double the number of dynamic instructions for a program, incurring significant performance and power penalties.

Symptom-based detection: As mentioned in previous sections, Wang et al. was the first to exploit anomalous microarchitectural behavior to detect the presence of a fault. Their light-weight approach for detecting soft errors, ReStore [109, 111], leveraged symptoms including memory exceptions, branch mispredicts, and cache misses. The concept of

anomaly detection has been further explored by Racunas et al. [69] who proposed verifying data value ranges and data bit invariants. Lastly, Li et al. [48] extended symptom-based fault coverage and applied it to detecting and diagnosing *permanent* hardware faults. The strength of symptom-based detection lies in its low-cost and ease of application. Unfortunately, the achievable fault coverage is limited and not appropriate for high error-rate scenarios.

Register file protection schemes: The register file holds a significant portion of program state. Consequently, error-free execution of a program cannot be accomplished without protecting it against faults. Just as main memory can be augmented with ECC, register file contents can also be protected by applying ECC. This process can be further optimized by protecting only live program variables, which usually occupy only a fraction of the register file. Solutions like the one presented by Montesinos et al. [58] builds upon this insight and only maintains ECC for those registers most likely to contain live values. Similarly Blome et al. [17] proposes a register value cache that holds duplicates of live register values. It is important to note that these schemes in general can only detect faults that occur after *valid* data has been written back to the register file. In contrast, Shoestring can also detect faults in other parts of the datapath that corrupt instruction output before it is written back to the register file or correction codes have been properly generated.

4.7 Summary

If technology scaling continues to exacerbate the challenges posed by transient faults, the research community cannot remain focused only on ultra-high reliability systems. We

must devote efforts also to developing new innovative solutions for mainstream commodity processors. This chapter introduces Shoestring, a transparent, software-based reliability solution that leverages both symptom-based detection as well as selective instruction duplication to minimize user-visible failures induced by soft errors. For a total performance penalty of 15.8%, Shoestring can cover an additional 33.9% of faults undetected by a conventional symptom-based scheme. Allowing just 1.6% of faults to manifest as user-visible data corruption, Shoestring is a cost-effective means of providing acceptable soft error resilience at a cost that the average commodity system can afford.

CHAPTER V

Encore: Low-cost, Fine-grained Transient Fault Recovery

Shoestring assumed that the presence of hardware speculation support could be easily repurposed to recover from transient faults. However, we acknowledge that the presence of this speculative hardware may not *necessarily* exist in systems with less aggressive performance targets. To accommodate processors with less robust hardware support we developed Encore, an alternative rollback recovery mechanism tailored for these lower-cost systems without support for speculative rollback recovery.

5.1 Introduction

Traditionally, architects have designed systems that would take periodic checkpoints of processor and memory state. In the event of a soft error the system could rollback to an existing, fault-free snapshot and continue execution (rollback recovery). These highly robust fault recovery solutions have historically also relied on some form of modular redundancy to provide the necessary detection capabilities. Available in spatial and temporal variants, modular redundancy generally involved redundant execution (either on separate hardware

or in separate software contexts) followed by detailed comparisons that would identify the presence of a fault [14, 95, 80, 74, 59]. However, the resultant overheads of these coupled detection and recovery schemes, a large component of which was the cost of creating checkpoints, usually relegated their use to to high-end, enterprise systems [28].

Given the rise of processor reliability as a first-order design constraint, even in lower-end commodity processors, there has been a growing interest in low-cost, non-intrusive mechanisms for transient fault detection. Many of these new proposals have been able to maintain low runtime overheads by sacrificing a small degree of reliability, focusing primarily on addressing the bulk of faults that are relatively inexpensive to detect [111, 48, 34]. However, these techniques [111, 34] have also tended to assume that existing hardware provided rollback recovery, arguing that such hardware would already be needed to support performance speculation. Although this argument may hold for aggressive out-of-order processors, such hardware support is not present in the majority of low-end commodity systems.

With that in mind we propose, *Encore*, a software-only solution that seeks to provide probabilistic (best effort) rollback recovery capabilities at minimal costs. Encore was developed to complement emerging probabilistic detection techniques, enabling them to be deployed in commodity systems without native hardware support for rollback recovery. As an automated, compiler-driven technique, Encore is able to utilize programmable heuristics that allow the end-user to dial in the desired degree of fault-tolerance and therefore only incur as much runtime overhead as they are able to budget. Encore can achieve this behavior by mimicking the same checkpoint, rollback, and re-execute model used by earlier enterprise systems. However, rather than performing, full-system, heavyweight checkpoints,

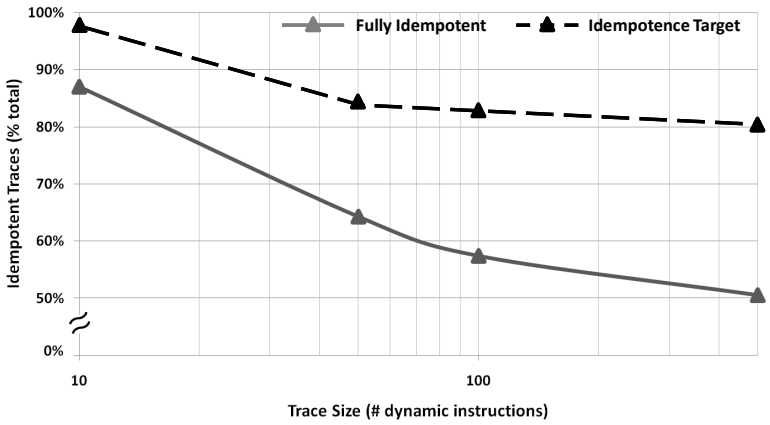


Figure 5.1: *Percentage of dynamic instruction traces that are inherently idempotent as a function of size. The execution traces were extracted from an assortment of SPEC2K and Mediabench workloads. The* Idempotence Target *curve illustrates Encore's goal of exposing, and exploiting, even greater degrees of idempotence through intelligent compiler analysis and transformations.*

Encore is able to exploit the *statistically idempotent* property of applications to reduce, and in certain situations nearly completely eliminate the overheads required to supply rollback recovery.

At a high-level, an idempotent region of code is simply one that can be re-executed multiple times and still produces the same, correct result. In the context of rollback recovery, this means that at least to the first order, a fault occurring within an idempotent piece of code can be recovered from without any overhead for checkpointing state. This typically means that there cannot exist any paths through the region that can read, modify, and then write to the same (or overlapping) location(s), i.e. no write after read (WAR) dependencies.

To better understand the extent of idempotent code present in an application, Figure 5.1 shows the distribution of idempotent execution traces across a set of desktop and media benchmarks. Results are shown as a function of the trace size (dynamic instruction length). The surprisingly large percentage of naturally occurring idempotent regions of execution seen in Figure 5.1 is what initially encouraged the development of Encore. To the first order,

the code regions corresponding to the traces that were identified as idempotent could be easily instrumented for rollback recovery with almost no impact on runtime performance. It is important to point out however, that although there is plenty of opportunity present, only a few of these regions actually span an entire function. Most are spread throughout the application, making manual inspection to identify them impractical.

This is not entirely unexpected since with more instructions comes the greater chance that there exists some sequence of instructions that violate the WAR constraints required to maintain idempotence. This intuition is reinforced by the data which exhibits a sharp drop in the likelihood of being idempotent when moving from traces with just a handful of instructions to those with 50 or more. Lastly, it is also interesting to note that for the traces that do not exhibit full idempotence, many tend to be *nearly* idempotent, i.e., containing only a few offending instructions. Furthermore, these instructions often only occur along statistically unlikely paths. Encore seeks to expose even greater amounts of statistical idempotence by recognizing these properties of application behavior (*Idempotence Target* in Figure 5.1).

To make exploiting program idempotence feasible, this chapter proposes techniques to automate the analysis and instrumentation within compiler optimization passes. We present the algorithms and heuristics developed that enable Encore to carefully partition application code into fine-grained regions with favorable idempotence behavior, and then to instrument them for rollback-recovery. By recognizing the statistically idempotent structure naturally present in many applications, Encore can transparently provide rollback recovery on commodity systems at prices that they can afford. The contributions of this chapter are as follows:

- We demonstrate how low-cost transient fault recovery can be achieved for commodity systems without hardware support for aggressive performance speculation.
- We develop new compiler algorithms and heuristics for
  - Automatically identifying candidate idempotent regions in generalized code regions with support for cycles.
  - Selectively trading off recoverability with cost by performing code transformations that leverage application profiling statistics.
- We evaluate and analyze the performance of Encore's ability to recover from transient faults with full-system simulations across a diverse set of representative workloads.

5.2 Recovering from Transient Faults

Transient fault tolerance requires the ability to *detect* and subsequently *recover* from soft error events. There is no shortage of examples in the literature that address each of these tasks (see Section 5.6). However, as we have already indicated, recent progress in achieving low-cost probabilistic transient fault detection has not been accompanied by similar advances in fault recovery. Encore, and the remainder of this chapter, is concerned with being able to rollback and recover from a transient fault once it has already been detected by a low-cost, low-latency solution like ReStore [111] or Shoestring [34].

Traditional high-reliability systems have chiefly relied upon heavy-weight, full-system checkpointing mechanisms to support rollback and recovery. Some high-level characteristics of these traditional techniques are highlighted in Table 5.1. Compared to these con-

ventional methods, Encore provides recoverability at much finer-granularities without any specialized hardware support. Although it cannot provide guaranteed recovery, the probabilistic nature of Encore allows it to be applied to commodity hardware at dramatically lower costs (in terms of runtime performance and memory footprint).

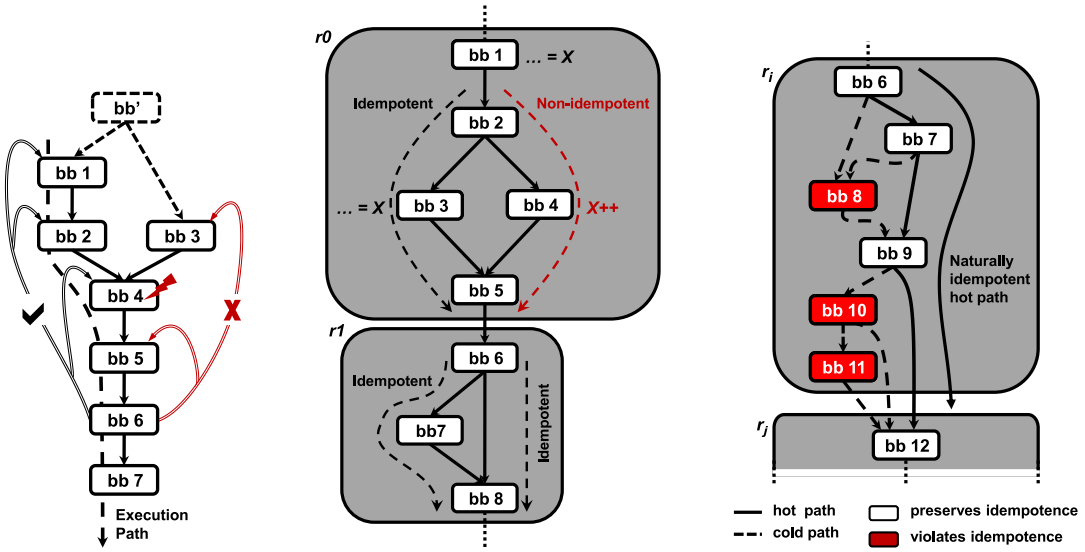
Table 5.1: *Comparison with conventional checkpointing schemes.*

Attributes	**Enterprise Recovery** [28, 61, 29, 95]	**Architectural Recovery** [94, 68]	**Encore**
Interval Length	~hours	100-500K instrs.	100-1000 instrs.
Storage Space	0.5 - 1 GB	0.5 - 1 MB	~10 - 100 B
Checkpoint Time	~minutes	~ms	~ns
Scope	Full System	Processor	Processor
Guaranteed Recovery	Yes	Yes	No
Extra Hardware	Sometimes	Yes	No

5.2.1 Recovery with Fine-grained Re-execution

At the high-level, one of the simplest ways to recover from a transient fault is by re-executing the application from a location far enough back along the control flow graph (CFG) so as to correctly reproduce the data that was corrupted by the fault. With this seemingly straight-forward maneuver, the effects of all but the most insidious transient faults can be completely eliminated. This, of course, assumes that during the initial execution no WAR dependencies overwrote state that could lead to erroneous behavior upon re-execution.

Note, that employing this form of fault tolerance requires, in addition to a detection mechanism, the ability to identify the location from which to initiate re-execution, i.e., deciding where the code should rollback to in the event of a fault. Ideally the system would rollback to point just before the fault site and no further. This would ensure correct forward



(a) Rollback options **(b)** Significance of idempotence **(c)** Significance of hot paths

Figure 5.2: *Fine-grained transient fault recovery via rollback and re-execution. (a), (b), and (c) illustrate some of the challenges and opportunities that exist when leveraging fine-grained re-execution to achieve fault recovery. (a) enumerates potential rollback destinations that execution can be redirected to once a fault, striking at bb_4 is detected, at bb_6. Ideally bb_1 and bb_3 would share a common predecessor bb' that could serve as the rollback destination for all faults that are detected within the region. (b) highlights how idempotence violating instructions might constrain which code regions can actually be efficiently recovered. (c) depicts how otherwise non-idempotent regions can still frequently exhibit idempotent behavior along their hot paths. The region shown in (c) is taken from the CFG corresponding to the dominant hot function in* 175.vpr.

progress while also minimizing the amount of "wasted work," the amount of code that was uneccessarily re-executed. Correctly reasoning about this location requires **1) that you can precisely diagnose where the fault occurred and 2) that you can identify the original path of execution that lead to the fault site**. Although conceptually simple, merely ascertaining 2) without specialized hardware support would require costly, software-based dynamic control flow signature generation [113]. Yet, even having established the original path of execution, accurately locating the site of the initial fault still remains an expensive, if not altogether impractical task.

Figure 5.2a helps illustrate these two challenges in more detail. Basic blocks $bb_1 - bb_7$ form a small subgraph of code taken from a larger CFG. In this example a transient fault

corrupts an instruction inside basic block bb_4. When the fault is detected at bb_6, we are left with the difficult task of determining where to redirect control to safely rollback and recover. Basic blocks bb_1, bb_2, and bb_4 are all viable options that would lead to safe recovery. However, rolling back to bb_5 would not be far enough, while re-executing from bb_3 could actually lead to other undesirable behavior since it was not on the original execution path. Ultimately, identifying the optimal location to redirect control after a fault is detected requires dynamic information and is undecidable at compile-time.

Yet, if the subgraph in Figure 5.2a were part of a single-entry, multiple-exit (SEME) region, the decision could be made to conservatively rollback execution to the region entry block (the dominating header), in this case bb'. This would not only free Encore from having to account for which path of execution originally lead to bb_4, but in this example it would also ensure that execution was resumed far enough "back" to regenerate any data corrupted by the original fault. Despite uneccessarily re-executing some code, namely bb_1 and bb_2, this is still a more agreeable alternative to specialized hardware additions or expensive dynamic control flow tracking. Although this effectively resolves challenge 2), obviously in the general case, resuming execution at the top of SEME regions is only effective against faults that are detected within the same region that they occur. Fortunately, for large SEME regions the probability of a fault being detected after control has left the region is reduced. Details regarding how Encore attempts to form these large SEME regions are described in Section 5.3.3.

5.2.2 The Role of Idempotence

Figure 5.2b illustrates how recognizing idempotent regions can greatly reduce the overhead required to provide rollback recovery. Since idempotent regions by definition contain no WAR dependencies, they are attractive candidates for Encore's re-execution based fault recovery. In this example, since all paths through region r_1 are idempotent, it is more desirable than r_0, for which execution down the path containing bb_4 can be non-idempotent. Relying on conventional, full-system checkpointing schemes to ensure that a region like r_1 could be re-executed would be using a sledgehammer to crack the proverbial nut. Because the region is naturally idempotent, Encore can simply redirect all fault detection events initiated anywhere within the region to bb_6, the header of r_1. It is important to note here that although r_0 is not idempotent, if the increment of variable X in bb_4 were the only instruction violating idempotence then selectively checkpointing X prior to the increment would transform r_0 into a readily recoverable region. Small, cost-efficient transformations like these, described in greater detail in Section 5.3, are what enable Encore to achieve low-cost rollback recovery.

Lastly, Figure 5.2c shows an actual subgraph taken from *175.vpr*, a benchmark from the Spec2000 benchmark suite. It corresponds to a slice of the CFG from the function *try_swap*, which is the hottest function within the application (accounting for roughly half of its execution time). The details of the basic blocks and the surrounding CFG have been abstracted away for clarity. The shaded basic blocks, bb_8, bb_{10}, and bb_{11} are locations where the idempotence of the region can be violated. The code within these basic blocks is responsible for memory allocation to dynamic variables. Consequently, these are only executed the first

time *try_swap* is called. For the remaining invocations of *try_swap*, the path through basic blocks bb_6, bb_7, and bb_9 dominates the execution time of r_i. This suggests that although region r_i is not strictly speaking idempotent, it does exhibit idempotent *behavior* the vast majority of the time. This *probabilistic* idempotence is yet another property of applications that can Encore exploit to reduce its overheads.

Admittedly the notion of idempotence is not new. For example, Kim et al. [45] leveraged idempotent properties of inner loops in Fortran applications to minimize the instances of storage overflows in a speculative execution system. However, relying on this property for low-cost, transient fault recovery in a systematic fashion has not yet been fully addressed. A recent proposal by Kruijf et al. [26] resorted to manually inspecting and modifying source code to take advantage of the function-wide idempotence and fault tolerant properties of multimedia and data-mining applications. Although their work is in the same spirit as Encore, and a great step in the right direction, by utilizing domain/application-specific algorithmic knowledge to identify and condition candidate functions significantly limits the applicability of the approach. **Establishing a generalized methodology for exploiting fine-grained, often statistical, idempotence to enable low-cost rollback recovery was the purpose of developing Encore.** The remainder of this chapter will address the algorithms and heuristics formulated to realize this goal.

5.3 Encore

Achieving low-cost transient fault recovery involves identifying naturally occurring regions of code that are amenable to re-execution, and judiciously sacrificing reliability to

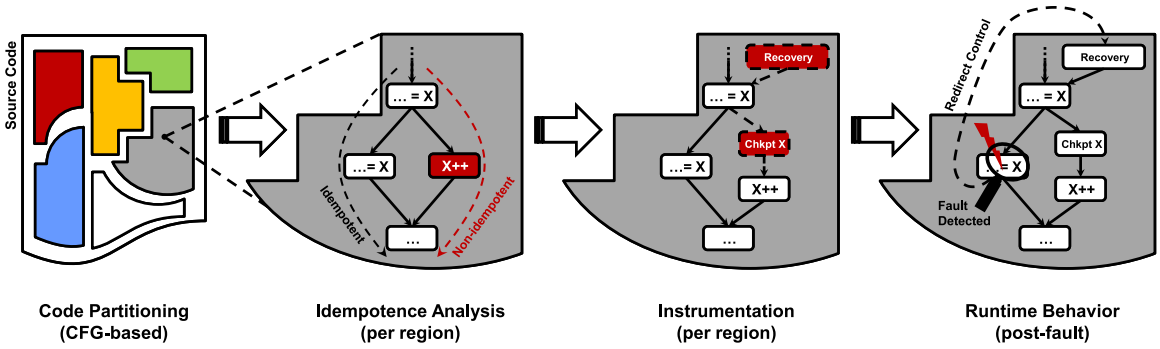


Figure 5.3: *High-level Encore vision. At compile-time, application code is partitioned into SEME regions that are subsequently analyzed and instrumented to enable low-cost rollback recovery from transient faults. Flexible heuristics enable Encore to refine the partitioning and instrumentation passes, customizing their behavior to achieve the desired tradeoff between reliability and performance overheads.*

maintain low overheads. Figure 5.3 illustrates the different high-level components of the Encore vision. Encore is designed as a series of compiler passes that analyzes, refines, and ultimately instruments the code with rollback recovery "hooks" that are coupled with a detection mechanism at runtime. To start with, the application source code (specifically the CFG) is initially partitioned into SEME regions. These regions are then analyzed to determine their idempotence properties (Section 5.3.1). The results of this analysis are then used to instrument the regions for rollback recovery (Section 5.3.2) as well as refine the initial region partitioning (Section 5.3.3) using heuristics developed to maximize rollback coverage while maintaining acceptable overheads (Section 5.3.4). Lastly, when faults are detected at runtime, execution is redirected to a recovery block that restores any non-idempotent state from lightweight checkpoints before releasing control to the header block of the corresponding region.

5.3.1 Identifying Inherent Idempotence

Before the discussion proceeds any further, to help avoid any ambiguity that may arise, a few terms that will be used throughout this chapter are explicitly defined below.

Region: in its unqualified form this will refer to SEME regions, a subgraph of basic blocks connected in the program CFG that contains a single (entry) block that dominates all other blocks and zero or more exiting blocks (basic blocks with branches to outside the region).

Reachable Store [1]: at a given point p (i.e., basic block) in the CFG, a *reachable store*, relative to p, is a store instruction that could potentially execute after control has passed through p.

Guarded Address: with respect to a given point p, a *guarded address* is one that is *guaranteed* to be overwritten by a store instruction prior to reaching p, along all possible paths to p.

Exposed Address: with respect to a given point p, an *exposed address* is an address that *may* be referenced by an unguarded load prior to reaching p. A load l is guarded if, and only if, the address referenced by l is already a guarded address with respect to the location of l.

Inherent Idempotence: a property of a region indicating that the region contains no WAR sequences to the same address that could prevent it from being safely re-executed during rollback recovery without undesired side-effects.

[1] This should not to be confused with the concept of reaching definitions commonly used in dataflow analysis.

These definitions, and the text throughout this chapter, only make reference to load and store instructions. This is done simply in an effort to improve readability. In reality, all instructions that can potentially reference and/or modify memory are considered during the idempotence analysis. Additionally, register state is also initially ignored in the analysis and will be treated separately in Section 5.3.2.

5.3.1.1 Path Insensitive Analysis

Determining the idempotence of a region, r, begins by generating the region-wide *reachable store* ($\mathbb{RS}$), *guarded address* ($\mathbb{GA}$) , and *exposed address* ($\mathbb{EA}$) sets for all basic blocks $bb_i \in r$. This is done by performing multiple post-order traversals of the region's CFG. For the time being, the details surrounding these regions will be ignored with the exception of saying that they are limited to SEME subgraphs of basic blocks. Initially the discussion will also be limited to acyclic regions. Cycles (i.e., loops) will be incorporated in Section 5.3.1.2 once the initial acyclic algorithm has been described.

The initial post-order traversal begins from the entry block to the region. As each basic block (bb_i) is encountered, Equation 5.1 is used to update the corresponding reachable store set, $\mathbb{RS}_{bb_i}$. Next, all edges in r are reversed and multiple post-order traversals are performed on this new subgraph starting from each of the region's exiting blocks. As each basic block, bb_i, is encountered during these "reverse" post-order traversals, Equations 5.2 and 5.3 are used to update the corresponding guarded address and exposed address sets. Note that the guarded address set, $\mathbb{GA}_{bb_i}$, must be updated before the exposed address set, $\mathbb{EA}_{bb_i}$.

Furthermore, the set subtraction operation, "−", used in these and subsequent equations is supplied with standard, conservative, static memory alias analysis techniques [2].

$$\mathbb{RS}_{bb_i} = \bigcup_{\forall bb_j \in \mathbb{C}_{bb_i}} \left(\mathbb{RS}_{bb_j} \cup \mathbb{AS}_{bb_j} \right) \bigcup \mathbb{AS}_{bb_i} \tag{5.1}$$

where $\mathbb{RS}_{bb_i}$ is the set of reachable stores at bb_i; $\mathbb{C}_{bb_i}$ is the set of bb_i's children; and $\mathbb{AS}_{bb_i}$ is the set of all stores within bb_i itself.

$$\mathbb{GA}_{bb_i} = \bigcap_{\forall bb_j \in \mathbb{C}_{bb_i}} \left(\mathbb{GA}_{bb_j} \cup \mathbb{AS}_{bb_j} \right) \tag{5.2}$$

$$\mathbb{EA}_{bb_i} = \left(\bigcup_{\forall bb_j \in \mathbb{C}_{bb_i}} \mathbb{EA}_{bb_j} \right) \bigcup \left(\mathbb{EA}_{bb_i}^{local} - \mathbb{GA}_{bb_i} \right) \tag{5.3}$$

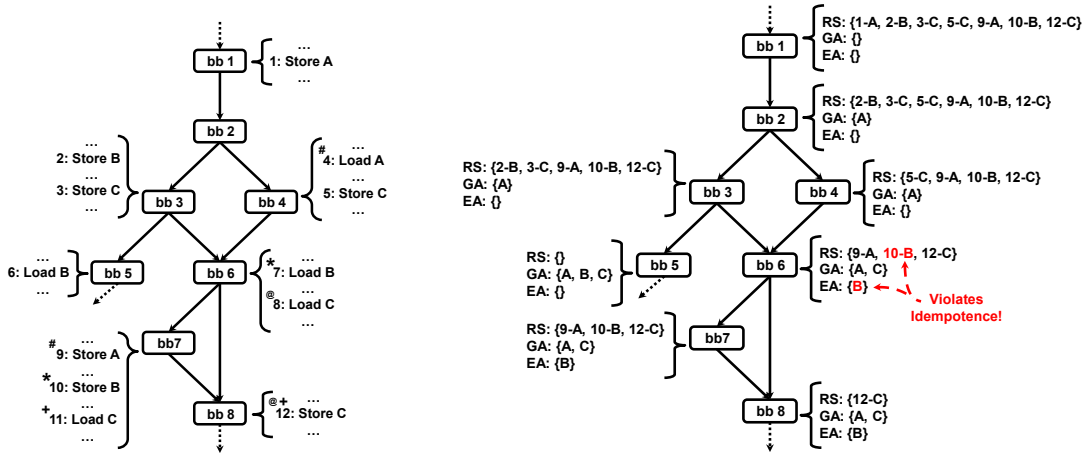
where $\mathbb{GA}_{bb_i}$ is the set of guarded addresses at bb_i; $\mathbb{EA}_{bb_i}$ is the set of exposed addresses at bb_i; $\mathbb{EA}_{bb_i}^{local}$ is the set of all addresses referenced by loads in bb_i that are not preceded by a store, also within bb_i, to the same address, effectively the set of *local* exposed addresses for bb_i.

Once all the basic blocks within the region have been processed, and the associated $\mathbb{RS}$, $\mathbb{GA}$, and $\mathbb{EA}$ sets have been generated, Equation 5.4 can be used to determine whether r is idempotent. It essentially checks if idempotence can be violated by executing a basic block bb_i along *any* possible path through r.

$$\begin{gathered} \text{Region } r \text{ is idempotent } \textbf{iff } I(bb_i) = \text{true}, \forall bb_i \in r \\ \text{where, } \quad I(bb_i) \begin{cases} \text{true}, & \textbf{iff } \mathbb{EA}_{bb_i} \cap \mathbb{RS}_{bb_i} = \emptyset \\ \text{false}, & \text{otherwise} \end{cases} \end{gathered} \tag{5.4}$$

Figure 5.4 illustrates how this path insensitive idempotence analysis is performed on a small example region. Figure 5.4 highlights the potential WAR dependencies (#,⋆,@,+) that exist between the relevant instructions 1-12. Figure 5.4b shows how the data struc-

[2] Extending Encore to use more aggressive dynamic memory profiling is a promising area of future work.



(a) A SEME region with four potential idempotence violating WAR dependencies: instructions 4 and 9 (#); 7 and 10 ($\star$); 8 and 12 (@); and 11 and 12 (+).

(b) Results of the Encore analysis that identifies the single dependency that actually requires checkpointing (instruction 10) to maintain idempotence. The number-letter pairs in the $\mathbb{RS}$ sets indicates the instruction number and destination address for the corresponding store.

Figure 5.4: *Example illustrating Encore's idempotence analysis. Only the relevant instructions within each basic block are shown in (a). (b) shows how the data structures in Equation 5.1 and Equations 5.2-5.3 are populated during the in-order traversal and reverse in-order traversal of the subgraph, respectively.*

tures in Equation 5.1 and Equations 5.2-5.3 are populated during the in-order traversal and reverse in-order traversal of the subgraph, respectively. Although there are four WAR dependencies that exist within this region, Encore is able to single-out the one dependence that can actually violate idempotence during runtime, the dependency between instructions 7 and 10 ($\star$).

Admittedly, identifying idempotence in this manner leads to conservative answers. Equation 5.4 does not account for correlations among branches between basic blocks and consequently may categorize regions as non-idempotent because of paths that can never be realized given the design of the application. However, augmenting any compiler analysis with path sensitive information is generally considered an intractable problem [32]. Nevertheless despite this limitation, the algorithm proposed here is efficient, scalable, and sufficiently accurate.

5.3.1.2 Incorporating Cycles

Up to this point, the analysis has focused on acyclic regions. Introducing cycles can dramatically complicate issues. To help maintain the scalability of the analysis, loops within a region are treated in a hierarchical manner. Initially, prior to idempotence analysis, a conventional compiler pass ensures that all loops are in a canonical form [3] (i.e., single header block and no side-entries). Next, whenever the boundaries of loops are encountered (header blocks during the forward post-order traversals and exiting blocks during the reverse post-order traversals) no attempt is made to enter the body of the loop. Instead, previously generated meta-information for each loop that summarizes the net impact of all the memory accesses within the loop is used to update idempotence data structures. This enables entire loops to be treated as if they were simply another basic block.

When analyzing regions containing cycles, all loops are processed first. If nested loops are present, they are analyzed from the inner-most loop outward. When processing an (inner-most) loop the constituent basic blocks can initially be analyzed as if they were just a simple acyclic region. The guarded address and exposed address sets for each basic block within the loop are generated as described in Section 5.3.1.1. However, given the cyclic nature of loops, effectively all stores are potentially reachable from any point within (possibly across iterations). Therefore, the set of reachable stores for each basic block within a loop l, $\mathbb{RS}^{l}_{bb_i}$, is equivalent to the set of all stores within the loop, $\mathbb{AS}^{l}$. Defining $RS^{l}_{bb_i}$ in this fashion ensures that all cross-iteration WAR dependencies are accounted for. Once the loop-wide reachable store, guarded address, and exposed address sets have been

[3] Not all cycles within a CFG can be converted into a canonical form. In these rare cases, Encore does not instrument the parent region for rollback recovery.

generated for all basic block within the loop, the loop can be treated as any other region and idempotence can be assessed using Equation 5.4. Once loop idempotence has been determined, the next step is to generate the meta-information associated with the loop.

The goal of loop-wide meta-information is to capture and expose loop-wide memory side-effects to simplify subsequent region analysis. This allows the entire loop itself to be treated effectively as a simple basic block. The contents of this data structure are enumerated below and are used in an analogous fashion to their basic block counterparts.

Loop-wide reachable stores, $\mathbb{RS}_{l_i}$: the set of all stores that could potentially execute if control ever enters loop l_i. The cyclic nature of the loop ensures that $\mathbb{RS}_{l_i} = \mathbb{RS}^{l_i}_{header} = \mathbb{AS}^{l_i}$, where $\mathbb{RS}^{l_i}_{header}$ is the set of reachable stores for the loop header and $\mathbb{AS}^{l_i}$ is the set of all stores within the l_i.

Loop-wide guarded addresses, $\mathbb{GA}_{l_i}$: the set of all addresses that are that are guaranteed to be overwritten if and when loop l_i is executed. Since loops can have multiple exiting blocks this is effectively the intersection of all guarded addresses across all exiting blocks of l_i. In other words $\mathbb{DS}_{l_i} = \bigcap_{\forall bb_i \in \mathbb{X}_{l_i}} \mathbb{GA}_{bb_i}$, where $\mathbb{X}_{l_i}$ is the set of exiting blocks for l_i.

Loop-wide exposed addresses, $\mathbb{EA}_{l_i}$: analogous to the definition of local exposed addresses for basic blocks, the exposed address set for l_i is the set of all addresses that may be referenced by an unguarded load along all possible paths through l_i. This is equivalent to the union of the $\mathbb{EA}$ sets across all the exit blocks, $\mathbb{EA}_{l_i} = \bigcup_{\forall bb_i \in \mathbb{X}_{l_i}} \mathbb{EA}_{bb_i}$

With the loop-wide meta-data populated for all loops within the CFG, analysis of any arbitrary region can proceed and treat loops hierarchically as just another basic block. The region traversals simply "step-over" loops whenever they are encountered and update idempotence data structures with the loop-wide meta-data.

5.3.2 Instrumentation

Once the idempotence of the various regions within an application has been determined, the next step is to identify whether inherently non-idempotent regions can be efficiently (with low runtime overheads) transformed into idempotent regions. For Encore, this transformation is achieved by instrumenting offending non-idempotent regions with instructions to checkpoint state that may otherwise be overwritten upon re-execution.

While performing the idempotence checks in Section 5.3.1, all offending stores that violate Equation 5.4 are recorded in a checkpoint set, $\mathbb{CP}$, associated with every region. If Encore decides to enable recovery (see Section 5.3.4), on a non-idempotent region, r_i, it will proceed to instrument each store, $s \in \mathbb{CP}$, with checkpointing instructions that checkpoints the data just prior to s. Additionally, in order to ensure that no WAR register dependencies violate idempotence, all live-in (with respect to r_i) registers that are overwritten within r_i are also checkpointed upon entering the region. The identification of register live-in values is a standard analysis in modern compilers and is omitted due to space constraints. Given the small amount (see Section 5.5) of storage required the checkpointed data is placed in a specially reserved region of the stack.

After instrumenting a region with the necessary checkpointing instructions, all that remains is to create a *recovery block*–the destination of all rollbacks, initiated if and when

a fault is detected within the region. Encore instruments the header of each region with a simple store that updates a dedicated memory location with the address of the corresponding recovery block each time control enters the region. Existing low-cost, software-based detection schemes [111, 34] can be easily modified to redirect control to the address stored in this reserved memory location when a fault is detected.

Within the recovery block, all the previously checkpointed state (registers and memory) are restored before redirecting control back to the region header. Although this additional instrumentation also contributes to runtime overheads, it is only executed upon the detection of a transient fault. In fact, the conditional rollback to the recovery block can also be amortized with the cost of the detection scheme. Further optimizations for reducing the overhead of this selective checkpointing are described in Section 5.3.4.

5.3.3 Region Formation

Having discussed how idempotence is analyzed (and enforced if necessary), we can now discuss how the CFG is actually is initially partitioned, and subsequently refined, into these segments. Candidate region formation is done in Encore by building upon traditional interval analysis [3]. In general an interval, as defined by Aho et al., is essentially a loop plus acyclic "tails" that dangle from the blocks within the loop. In practice the initial loop at the "top" of the interval may not exist (i.e., an interval can simply be a small SEME subgraph that shares a single dominating header node).

Since it is a standard pass within most modern compilers, this section omits the details of how this *initial* partitioning is achieved and focuses on the subsequent refinement steps. However there are two properties of this partitioning that are important to keep in mind,

1) All intervals are by definition SEME regions and 2) interval partitioning can be applied recursively.

The first property of interval partitioning greatly simplifies the process of recovery. By ensuring that all regions are SEME, Encore can avoid the costly task of tracking dynamic execution paths (see Section 5.2.1). This property is what allows Encore to safely insert the recovery block described in Section 5.3.2 just before the region's header. Irrespective of which path lead to the actual fault site, redirecting control to this recovery block will ensure that it can be corrected.

The second property suggests that once a CFG is partitioned into intervals, the intervals themselves form an *interval graph* that can also be partitioned into intervals. Encore exploits the second property of interval partitioning to create candidate regions with varying sizes. By controlling the size of the regions, Encore is able to effectively manage the trade-off between fault tolerance and performance overhead. Generally speaking, the larger the region that Encore attempts to recover from, the greater the likelihood that the region is not inherently idempotent. Recall that non-idempotent regions require instrumentation to enable safe re-execution, which contributes to the overall runtime overhead. On the other hand, the larger the region, the more likely that a transient fault striking within the region will be detected before control exits the region and the fault is no longer recoverable. Section 5.3.4.2 will discuss how heuristics are used to identify the appropriate region size given a budget for acceptable performance overhead.

At this point, one might contend that merging two regions, r_i and r_j to form a larger region r' (with r_i preceding r_j) may not necessarily incur additional costs to enforce idempotence within r'. In fact, if r_j were non-idempotent, the fact that it is preceded by r_i

could actually reduce the amount of checkpointing required in r_j if the idempotence violating instructions within r_j only referenced locations within the guarded address set for r_i. Although in principle this is correct, in practice these scenarios are rare and for the majority of cases, fusing regions together was not an effective means of *reducing* performance overheads.

5.3.4 Encore Heuristics

This section will focus on the heuristics used to glean the best reliability versus performance trade-offs from Encore. First we discuss how profiling information can be used to statistically prune basic blocks from the idempotence analysis followed by the heuristic used to identify which regions are chosen as candidates for rollback recovery.

5.3.4.1 Relaxing Idempotence Criteria

Since Encore is intended to supply probabilistic rollback recovery for non-mission critical systems, one opportunity for optimization is to leverage application profiling. Unlike conventional techniques targeting ultra-reliable systems that must provide guarantees on recoverability, Encore is free of such constraints and is at liberty to to utilize profile-based, not necessarily provable, analysis.

Presented with this flexibility the algorithm described in Section 5.3.1 can selectively ignore any basic blocks that do not meet a certain "liveness" criteria. As previously mentioned, the idempotence determination made by Equation 5.4 is necessarily conservative since it accounts for all paths through the region. By exploiting profiling information, Encore can now exclude basic blocks that are along paths that have low probabilities of being

traversed when updating $\mathbb{RS}$, $\mathbb{GA}$, and $\mathbb{EA}$ sets for each basic block. More formally, this means that Equations 5.1, 5.2, and 5.3 can be re-formulated limiting the union and intersection operations, which originally operated over all the children of a basic block, $\mathbb{C}_{bb_i}$, to a subset set of children $\mathbb{C}'_{bb_i}$ where the *dynamically-dead* children have been pruned away. The degree to which Encore filters these rarely executed basic blocks from its idempotence analysis is controlled by a heuristic parameter P_{min}. Any basic block with an execution probability less than P_{min} is selectively ignored.

5.3.4.2 Region Selection

Another opportunity for trading off reliability for performance is in the area of region selection. Encore can selectively decide **1) which regions should be instrumented for recovery (Section 5.3.2) as well as 2) when to terminate the process of merging existing intervals to form larger regions (Section 5.3.3)**. Exposing the heuristic parameters that control these decisions allows Encore to be customized by system designers.

Determining whether protecting a region is actually a profitable endeavor, is fairly straightforward. For inherently idempotent regions, the answer is almost always yes. The cost of updating the address for the current recovery block is negligible for all but the smallest possible regions. However, for small, non-idempotent code portions, the overhead incurred to preserve idempotence can potentially make it more attractive to simply concede fault coverage for those regions. To account for this possibility, only regions that have reasonable cost-to-coverage ratios are instrumented for selective checkpointing and rollback recovery. In other words $Coverage(r_i)/Cost(r_i) > \gamma$ must be satisfied for every candidate region, where γ is a heuristic threshold. The length of the hotpath through r_i is used as a

compile-time surrogate for coverage, while the ratio of checkpointing instructions required to the number of total instructions within the hotpath is used as an estimate for cost.

Although some regions may initially be undesirable, Encore has the ability to merge adjacent regions to form larger, possibly more attractive candidates. Since merging regions has the potential to incur additional checkpointing instructions, it is only performed if the additional cost $\Delta Cost$ is offset by the improvement in overall reliability. Given two regions, r_i and r_j, that are to be fused into r', this $\Delta Coverage$ is defined as $\Delta Coverage(r') = Coverage(r')/Max(Coverage(r_i), Coverage(r_j))$.

At a given cost, this definition for $\Delta Coverage$ ensures that fusing two similarly sized regions, which earns more cost-effective returns in terms of improved reliability, is preferred over merging a large and a small region. Ultimately only when $\Delta Coverage/\Delta Cost > \eta$ does Encore consider merging existing regions. Small values of η predisposes the system to try and create the larger regions in pursuit of greater reliability while larger η's shift the focus toward minimizing performance overheads.

5.4 Experimental Methodology

As with all reliability schemes dealing with transient faults, an ideal evaluation of Encore would involve electron beam experiments on real hardware running real-world applications. Yet, given limited resources an acceptable alternative has been statistical fault injection (SFI) on detailed system models (architectural, microarchitectural, RTL, etc.). Statistics related to fault masking, and to a lesser extent fault detection, can be highly dependent on the details of the underlying hardware. Consequently, for the full system

results shown in Section 5.5.4 fault masking was determined by a series of Monte Carlo experiments that injected faults into a low-level Verilog model of an ARM926 embedded processor. Transient faults were injected into state elements and combinational logic and the overall average hardware masking rate was quantified.

However in contrast, the more important figure of merit for evaluating Encore, the amount of application code that can be cheaply re-executed, is far more sensitive to program structure and to some degree is (micro)architecturally neutral. The remaining details of the experimental methodology and the analytical model developed to evaluate Encore are described below.

5.4.1 Compilation Framework

The compiler analysis and instrumentation passes described in Section 5.3 were implemented in the LLVM compiler. An assortment of Spec2000 integer (SPEC2K-INT), Spec2000 floating point (SPEC2K-FP), and Mediabench applications serve as the representative workloads for our experiments. All applications were compiled with standard -O3 optimizations.

5.4.2 Recoverability Coverage Model

As previously stated, Encore only targets the *recovery* aspect of processor reliability. Within this context we define *recoverability coverage* as the percentage of application code that can be safely re-executed in the presence of a fault. In the case of Encore, this coverage is effectively equivalent to the percentage of execution time that is spent within dynamic

code regions that are inherently idempotent or have been instrumented to preserve idempotence.

5.4.2.1 Impact of Detection Latency

Assume that the hot path through region r consists of instructions $i_0, i_1, ..., i_n$. If a fault corrupts the output of i_s (where $0 \leq s \leq n$) and the detection latency for the system is l instructions, Encore can recover from this fault if $s + l < n$. To account for the detection latency of the system we calculate a latency scaling factor α according to Equation 5.5.

$$\begin{aligned} \alpha_{r_i} &= Pr(s + l < n),\ \forall s \in [0, n], \forall l \in [0, D_{max}] \\ &= \int_0^n \int_0^s f(l)g(s)dlds \qquad \text{where,} \end{aligned} \tag{5.5}$$

α_{r_i} : is the scaling factor associated with region r_i that accounts for detection latency.
n : is the number of (dynamic) instructions along the hot path through region r_i.
s : is a random variable, distributed over the interval $[0, n]$, representing the instruction (number) at which a transient fault occurs.
l : is a random variable, distributed over the interval $[0, D_{max}]$, representing the detection latency of a system with a maximum latency of D_{max}, measured in terms of instructions.
$Pr(s + l < n)$: the probability that a fault at instruction s is detected inside the boundary of region r_i.
$f(l)$: is the probability density function corresponding to the the detection latency of the system.
$g(s)$: is the probability density function corresponding to the fault sites within region r_i.

For the full-system fault coverage results presented in Section 5.5.4 we use an uniform distribution of fault sites, which assumes that every dynamic instruction over the course of an application's runtime has the same probability of being "struck" by a transient fault. This is consistent with the general body of reliability work that use uniform distributions to

guide the selection of fault locations and times during SFI simulations. Similarly, the full-system results in Section 5.5.4 also assumes a uniform distribution ([0,100] instructions) of fault detection latencies. This is consistent with the detection latencies exhibited by techniques like Shoestring [34], one of many recent proposals that exploits the anomalous software behavior that manifests in the wake of a soft error event. With these assumptions, Equation 5.5 can be re-written as Equation 5.6.

$$\alpha_{r_i} = \int_0^n \int_0^{min(s,D_{max})} \Big(\frac{1}{n}\Big)\Big(\frac{1}{D_{max}}\Big) dlds = \begin{cases} 1 - \frac{D_{max}}{2n}, & n \geq D_{max} \\ \\ \frac{n}{2D_{max}}, & n < D_{max} \end{cases} \tag{5.6}$$

5.4.3 Performance Modeling

The runtime performance overheads in Section 5.5.3 are presented in terms of dynamic instructions. The use of dynamic instructions may appear at first to be a less desirable alternative to running natively on a real machine and/or a microarchitectural simulator. However, this decision allows us to abstract away the details of the underlying hardware and present architecture-neutral results.

5.5 Evaluation and Analysis

This section presents the quantitative evidence demonstrating Encore's ability to provide affordable rollback recovery. For the data presented in this section values for γ and η (Section 5.3.4.2) were empirically derived for each application to target an acceptable maximum runtime overhead of $\sim 20\%$.

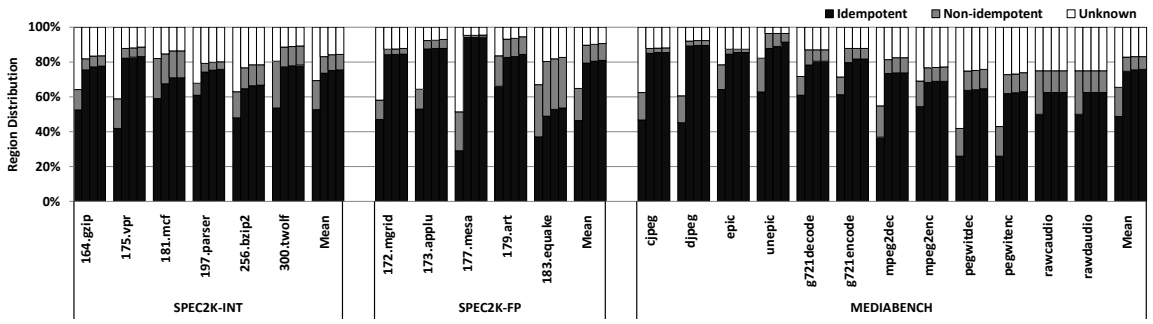


Figure 5.5: *Inherent region idempotence as a function of P_{min}. From left to right, the columns illustrate the fraction of regions within each application that is inherently idempotent for different values of $P_{min} \in \{\emptyset, 0.0, 0.1, 0.25\}$. With $P_{min} = \emptyset$, the left-most column for each application depicts the idempotence breakdown when no dynamically-dead code is pruned from the analysis. The* Unknown *segments correspond to portions of the application source code that could not be analyzed by Encore.*

5.5.1 Region Idempotence

Figure 5.5 examines the inherent idempotence of candidate recovery regions as a function of P_{min}. From left to right, the different columns for each application correspond to the idempotence for the different values of $P_{min} \in \{\emptyset, 0.0, 0.1, 0.25\}$. The different segments represent the fraction of regions that were identified to be inherently *idempotent*, *non-idempotent*, and *unknown*. Unknown regions contained code that Encore's compiler analysis was unable to process. This consisted of regions with calls to functions (mainly system and library function calls) for which relevant alias analysis information could not be easily obtained, preventing idempotence determinations.

Note that, as expected, the fraction of regions that are deemed idempotent grows as more dynamically-dead code is pruned (increasing values of P_{min}). Furthermore, nearly all of the benefit can be garnered by simply pruning the code that was *never* executed during profiling runs ($P_{min} = 0.0$). This suggests that a good portion of the instrumentation

optimizations described in Section 5.3 can be achieved without incurring any measurable risk. For the remainder of this section, all data presented is for $P_{min} = 0.0$.

Not surprisingly the SPEC2K-FP and Mediabench applications exhibit slightly better idempotence behavior than the SPEC2K-INT benchmarks. As suggested by Kruijf et al. [26], the multimedia and embedded-type codes typical of emerging applications tend to have fewer memory side-effects, great for idempotence. However, it is interesting to note that at least in terms of static code, on average, the extent of idempotence present across all three benchmark suites are comparable. It is encouraging to observe that even in control-heavy SPEC2K-INT applications, there is still a considerable fraction of code that is *inherently* idempotent. On average, across all applications, 49% of regions are inherently idempotent without pruning and 75% are idempotent with $P_{min} = 0.0$. This suggests that Encore would only have to insert minimal, if any, checkpointing instrumentation code for most applications to enforce idempotence.

5.5.2 Dynamic Execution Breakdown

Figure 5.6 takes a closer look at the execution of these workloads and presents the breakdown of execution time (calculated in terms of the percentage of total dynamic instructions) spent in different regions of the code. The segments labeled *w/ Encore Checkpointing* correspond to execution within regions that were non-idempotent but were selectively instrumented to preserve idempotence, while *w/o Encore Checkpointing* represents execution time spent in regions that were inherently non-idempotent but were too expensive to checkpoint. Execution time represented by the *w/o Encore Checkpointing* segment corresponds to lost recoverability coverage.

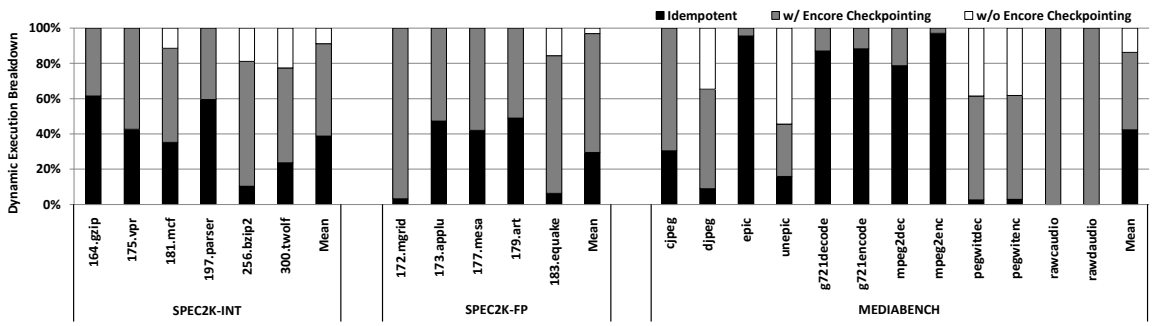


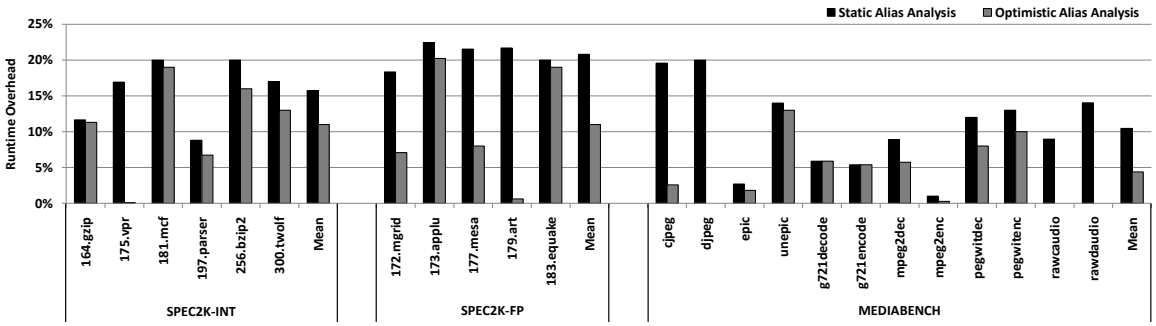
Figure 5.6: *Breakdown of dynamic execution time. For each application the stacks represent the fraction of execution time spent within regions of the code that were inherently idempotent, non-idempotent but instrumented with selective checkpointing by Encore, and non-idempotent but too costly to checkpoint.*

Despite having roughly the same amount of idempotent static code, the SPEC2K-FP and Mediabench workloads spent significantly more runtime within naturally idempotent and easily checkpointed code regions, i.e., Encore recoverable code.

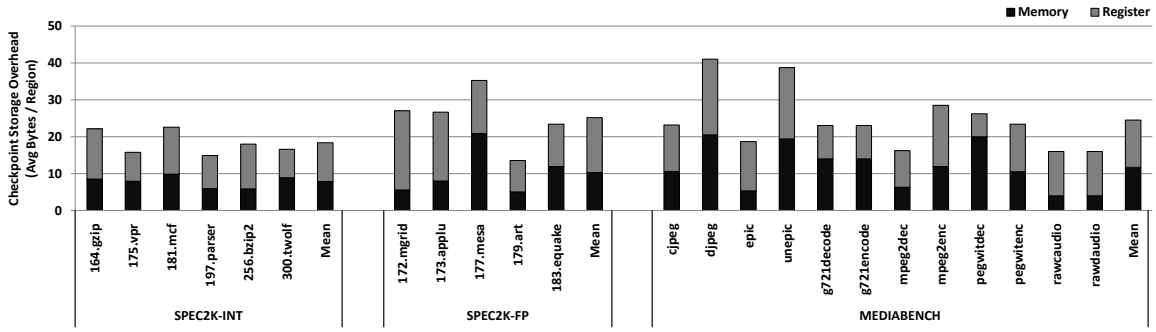
5.5.3 Overheads

Figure 5.7a reports the runtime overheads corresponding to the recoverability coverage results reported in Figure 5.6. The *Static Alias Analysis* bar shows the current runtime overheads for Encore while the *Optimistic Alias Analysis* bar provides an approximate lower-bound for future Encore designs that could utilize more robust alias analysis frameworks. Encore must currently checkpoint a significant number instructions because the limited alias analysis available to it cannot effectively disambiguate their addresses. Future, systems with more powerful, potentially dynamic, alias analysis could determine that a large fraction of these currently idempotence-violating instructions are in fact innocuous.

Nevertheless, even in its current form Encore only imposes a 14% runtime overhead, on average, across all benchmarks. Although Encore was given a 20% performance bud-



(a) Runtime performance overhead. The *Static Alias Analysis* bar reports the current runtime overheads for Encore while the *Optimistic Alias Analysis* bar provides an approximate lower-bound for future Encore designs that could utilize more robust (potentially dynamic) alias analysis frameworks.



(b) Overheads reported as the average number of bytes required per region to store checkpointing information. The stacked breakdowns highlight the contributions from memory and register checkpointing.

Figure 5.7: *Encore runtime and storage overheads.*

get, obviously not all workloads incurred this overhead. Some, like *172.mgrid*, *epic*, and *mpeg2enc* were able to instrument all regions for recovery without requiring the full performance budget. Others, like *181.mcf* and *177.mesa* were not able to meet (approach) the 20% target without incurring significant reductions in recoverability coverage.

Similarly, Figure 5.7b reports the storage overheads required to hold the selective checkpointing information generated by Encore. Note that for register checkpoints, the checkpointing information only consists of the register data, whereas for memory checkpoints both data and address must be stored to enable proper recovery. On average, Encore must only store 24 bytes of information per region, orders of magnitude less than the memory footprint of conventional, full-system checkpointing techniques.

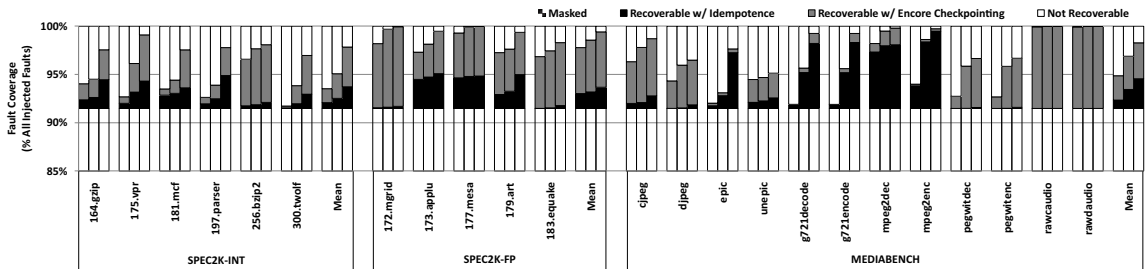


Figure 5.8: *Full-system fault coverage for a low-cost commodity system using Encore for rollback recovery and fault detection schemes with different latencies. From left to right, the columns represent the % of all transient fault events that can be effectively tolerated given a system with detection latencies of 1000, 100, and 10 instructions.*

5.5.4 Full-system Reliability

Lastly, Figure 5.8 examines the full-system fault tolerance that can be achieved by a commodity system augmented with Encore for rollback recovery and a Shoestring-like mechanism for fault detection. The *Masked* segments represent the fraction of transient faults that are naturally masked by the underlying hardware and do not require any additional intervention. As mentioned in Section 5.4, this masking rate was identified with Monte Carlo-based SFI experiments on a Verilog model of a representative low-end commodity processor [17].

In addition to the "free" fault coverage due to hardware masking, the fraction of faults the system can also recover from with Encore-enabled rollback recovery is represented by the *Recoverable w/ Idempotence* and *Recoverable w/ Encore Checkpointing* segments. Portions of the bars labeled *Not Recoverable* correspond to faults that either occurred within regions of the application code that Encore chose not to protect, or were the result of faults that were not detected before execution had already left the region containing the original fault site.

The different columns in Figure 5.8 correspond to the use of detection schemes with different latencies. From left to right, the columns represent the coverage for systems with detection latencies of 1000, 100, and 10 instructions. The middle column illustrates the coverage achievable for a system experiencing fault detection latencies consistent with existing techniques like Shoestring [34] and Restore [111]. The leftmost bar shows that Encore can even benefit systems with hardware speculation support. Since aggressive out-of-order machines typically only support rollback of 10-100 instructions, Encore could enhance the recoverability of these systems but supporting rollback even in cases where detection latencies reached 1000 instructions. Lastly, the rightmost bar presents the potential fault coverage that can be achieved in future systems with further constrained detection latencies.

Nevertheless, even with present day latencies, Encore can safely recover from 97% of faults, on average, across all benchmarks and nearly all faults for certain workloads like *172.mgrid*, *177.mesa*, *mpeg2dec*, and *rawcaudio*. Although these coverage results may seem less impressive when compared with the base masking rate of 91%, one must view these gains in the proper context. By supplying this 66% reduction in the number of transient events that can cause system failures, Encore can enable low-end commodity systems to meet reliability targets that may otherwise be out of reach.

5.6 Related Work

Transient fault tolerance requires two steps: 1) detecting the fault event and 2) recovering to an error-free state and resuming execution. Since Encore targets system recovery,

this section only contains a brief overview of fault detection solutions, while providing a more detailed discussion of previous efforts in fault recovery.

5.6.1 Fault Detection

There exists a large body of research addressing the challenge of fault detection [55, 34, 66, 71, 108, 80, 59, 75, 81]. These efforts can be broadly divided into four categories. First, there are solutions that utilize some form of spatial redundancy to execute multiple copies of an application simultaneously, periodically comparing results. Redundant multi-threading [80] and dual-core execution [91] are good examples from this class. The second category consists of solutions that exploit temporal redundancy, where the same work is re-executed on the same hardware resource. Compiler-based instruction duplication [75] and hardware-based selective replication [62] are well known techniques that fall into this group. Lately, a third class of techniques have emerged that rely on high-level software symptoms [108, 69, 81] to identify faults, sometimes with help of specialized detectors [55, 113]. Finally, there have also been recent proposals that formulate hybrid solutions [34, 71] combining multiple techniques to drive costs even lower while maintaining high detection coverage.

5.6.2 System Recovery

Once a fault is detected, the system must rollback in order to continue execution from a previous clean state. Recovery solutions are tasked with maintaining this clean state, and providing an interface to enable the rollback. The most popular category of recovery solutions are checkpoint based. In their simplest form, checkpoint-recovery solutions peri-

odically save off the entire system state, and revert to the most recent version in the event of a fault.

Enterprise-level Recovery. Traditionally, checkpoint-recovery solutions have been used in large-scale enterprise systems to guarantee the often touted "five-nines" of reliability. These systems, with 100-1000s of nodes, periodically suspend their program execution and take snapshots of the entire memory system, usually stored on globally accessible disks [28]. To maintain consistency, all the nodes in the system take their checkpoints simultaneously, often causing bottlenecks due to disk bandwidth limitations. In general these enterprise-level solutions are appropriate for their domain, but the cost of creating these checkpoints are prohibitively high for all but the most mission-critical systems.

Architectural Recovery. A cheaper alternative to taking a complete system snapshot is to log incremental changes to the system state. In the event of a failure, these changes can be unrolled as needed. SafetyNet [94] and ReVive [68] are two examples of such solutions. Although these log-based recovery solutions are scalable to more frequent checkpoints, and smaller intervals, the additional complexity and overheads introduced from potential hardware additions makes them less attractive for budget-wary commodity systems.

Opportunistic Recovery. This last category of recovery solutions may not technically be recovery schemes in the conventional sense. Work by Li and Yeung [50], subsequently reaffirmed by others [89, 83, 96, 26], recognized that not all applications, or even functions within an application, require the same degree of "correctness." Many, especially multimedia and embedded codes can naturally tolerate a non-trivial amount of errors. Li and Yeung

exploit this notion of application-level correctness by manually inserting checkpointing code that only saves the program counter, architectural register file, and stack state at the top of outer loops. Relax [26] takes this principle even further. Functions are manually instrumented with recovery blocks that are allowed to select between re-executing code, returning default values, or simply ignoring the faults depending on how such actions are expected to impact the "quality" of externally visible results. Encore shares the same basic principles with these other lightweight recovery solutions. However, this work is the first to present an automated (compiler-based, without manual inspection), generalized (beyond inner/outer loops and functions) solution for achieving selective rollback recovery. Furthermore, the application-level correctness notions [96] that existing works benefit from are complementary to Encore and can be supplied to the compilation framework to further enhance reliability.

5.7 Summary

Whether due to environmental phenomena or ambitious designs pushing the envelop of low power architectures, transient faults are re-emerging as a prominent reliability issue in modern computing. Yet despite this growing reliability concern, we would argue that instead of appropriating large transistor budgets (or processor cycles) to hedge against growing fault rates, system architects should embrace the high degree of fault tolerance that can be had simply by sacrificing provable guarantees. Such tradeoffs are the most attractive for low-end commodity and embedded markets, where systems often cannot afford to devote a substantial portion of their resources to anything other than actually perform-

ing useful computations. With the ability to recover from, on average, 97% of transient faults (when paired with existing detection mechanisms), Encore is poised as an attractive solution. Realizing this coverage at a modest 14% average performance overhead, it frees system designers to return their attention back to other aspects of the system architecture.

CHAPTER VI

Conclusion

In a world where consumer electronics permeate nearly every area of daily life, concerns over processor reliability will soon take center stage. Whether people are purchasing faster, more capable smartphones or the latest generation of tablets and laptops they will continue to demand a user experience that remains sheltered from the limitations of hardware reliability. Although the laws of physics will inevitably lead to more vulnerable transistors, microarchitects and system designers must develop innovative solutions that can keep emerging reliability threats at bay while imposing minimal user-visible overheads.

As our preliminary studies eventually evolved into the individual works presented within this thesis, we identified two fundamental insights that established the foundation of our work: 1) that the majority of consumer devices do not lie at either extreme of the reliability spectrum, necessitating the need for not just “low-cost” but genuinely affordable fault-tolerance, and 2) that with the appropriate analysis, the inherent computational patterns within programs can be leveraged to reap dramatic reductions in the cost of dependable computing.

In this dissertation we championed an approach to fault-tolerance that exploited the relaxed constraints of commodity devices in order to entertain ideas that have previously been overlooked by those in the high-reliability, mission-critical computing domains. Traditional fault-tolerance solutions are simply overdesigned for the average consumer who realistically does not expect "five-nines" of reliability from their devices. By sacrificing a few "nines", the user-centric, application-aware techniques described in this dissertation are able to provide a relatively transparent—in terms of additional hardware cost and performance degradation—layer of reliability that shields the end user from the negative effects of technology scaling.

The first half of this thesis embraced this philosophy and demonstrated that the risks posed by device wearout and permanent faults can be effectively addressed without resorting to traditional hardware overprovisioning. Instead of utilizing cold spares or intrusive circuit enhancements, Maestro relies on light-weight sensors like the WDU and the characteristic hardware and software heterogeneity present within chip multiprocessor environments. By introducing wearout-aware job scheduling algorithms we can achieve intra- and inter-chip wear leveling that significantly prolongs the effective life of the overall system, with virtually no impact on performance.

Next, we turned our attention to the more immediate threat of transient soft-errors. By leveraging efforts from two prominent areas of prior research, namely symptom-based fault detection and compiler-directed instruction duplication, we were able to produce an ultra-low cost, hybrid transient fault detection scheme, Shoestring. The flexibility of selective instruction duplication enabled us to compliment the basic fault-coverage supplied by more efficient, symptom-based detection methods. Intelligent compile-time analysis

allows Shoestring to apply instruction duplication sparingly, strategically protecting only the statistically-vulnerable portions of an application that are not inherently safeguarded by symptoms.

Lastly, having investigated fault-detection, our focus naturally turned toward transient fault recovery. With many recent proposals, including Shoestring, relying on existing hardware to provide fault recovery capabilities, a need emerged for a low-cost, software-only mechanism that could support recovery in commodity systems without native hardware for speculative rollback. With Encore we introduced a solution that appreciated the innate idempotent nature of many regions of program execution. Combining this observation with detailed program analysis, and a few choice compiler transformations, Encore is able to deliver respectable recoverability "coverage" without the specialized hardware or performance penalties of conventional, full-system checkpointing techniques.

By applying the law of diminishing marginal utility to fault-tolerant computing, the works presented in this thesis are able to break from tradition and advocate a new approach to looking at the reliability challenges facing future computer systems—one that dispenses with an all-or-nothing paradigm in favor of flexible engineering solutions that can target multiple design points with different cost-benefit tradeoffs.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] Gartner data systems conference, Dec. 2005. 11

[2] M. Agarwal, B. Paul, and S. Mitra. Circuit failure prediction and its application to transistor aging. In *Proc. of the 2007 IEEE VLSI Test Symposium*, Apr. 2007. 12

[3] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, 1986. 132

[4] E. R. Alliance. Online survey results: 2001 cost of downtime, 2001. 11

[5] Alpha. 21364 family, 2001. http://www.alphaprocessors.com/21364.htm. 50, 64

[6] A. Andrzejak, M. Arlitt, and J. Rolia. Bounding the resource savings of utility computing models, Dec. 2002. HP Laboratories, http://www.hpl.hp.com/techreports/2002/HPL-2002-339.html. 71

[7] J. S. S. T. Association. Failure mechanisms and models for semiconductor devices. Technical Report JEP122C, JEDEC Solid State Technology Association, Mar. 2006. 13, 14

[8] T. Austin. Diva: a reliable substrate for deep submicron microarchitecture design. In *Proc. of the 32nd Annual International Symposium on Microarchitecture*, pages 196–207, 1999. 39, 109

[9] T. Austin, D. Blaauw, T. Mudge, and K. Flautner. Making typical silicon matter with razor. *IEEE Computer*, 37(3):57–65, Mar. 2004. 41

[10] T. Austin, E. Larson, and D. Ernst. Simplescalar: An infrastructure for computer system modeling. *IEEE Transactions on Computers*, 35(2):59–67, Feb. 2002. 64

[11] W. Bartlett and L. Spainhower. Commercial fault tolerance: A tale of two systems. *IEEE Transactions on Dependable and Secure Computing*, 1(1):87–96, 2004. 7, 109

[12] M. Batty. Monitoring an exponential smoothing forecasting system. *Operational Research Quaterly*, 20(3):319–325, 1969. 29

[13] B. G. U. Berkeley. Bsim4 mosfet model, 2007. 16

[14] D. Bernick, B. Bruckert, P. D. Vigna, D. Garcia, R. Jardine, J. Klecka, and J. Smullen. Nonstop advanced architecture. In *International Conference on Dependable Systems and Networks*, pages 12–21, June 2005. 2, 7, 11, 76, 109, 115

[15] J. A. Blome, S. Feng, S. Gupta, and S. Mahlke. Online timing analysis for wearout detection. In *Proc. of the 2nd Workshop on Architectural Reliability*, pages 51–60, 2006. 12

[16] J. A. Blome, S. Feng, S. Gupta, and S. Mahlke. Self-calibrating online wearout detection. In *Proc. of the 40th Annual International Symposium on Microarchitecture*, pages 109–120, 2007. 55

[17] J. A. Blome, S. Gupta, S. Feng, S. Mahlke, and D. Bradley. Cost-efficient soft error protection for embedded microprocessors. In *Proc. of the 2006 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 421–431, 2006. 112, 144

[18] S. Borkar. Designing reliable systems from unreliable components: The challenges of transistor variability and degradation. *IEEE Micro*, 25(6):10–16, 2005. 10, 42, 79

[19] F. A. Bower, D. J. Sorin, and S. Ozev. A mechanism for online diagnosis of hard faults in microprocessors. In *Proc. of the 38th Annual International Symposium on Microarchitecture*, pages 197–208, 2005. 2, 39, 76

[20] D. Brooks and M. Martonosi. Dynamic thermal management for high-performance microprocessors. In *Proc. of the 7th International Symposium on High-Performance Computer Architecture*, pages 171–182, Jan. 2001. 48

[21] D. Brooks, V. Tiwari, and M. Martonosi. A framework for architectural-level power analysis and optimizations. In *Proc. of the 27th Annual International Symposium on Computer Architecture*, pages 83–94, June 2000. 64

[22] A. Cabe, Z. Qi, S. Wooters, T. Blalock, and M. Stan. Small embeddable nbti sensors (sens) for tracking on-chip performance decay. In *Proc. of the 2009 International*

Symposium on Quality of Electronic Design, Washington, DC, USA, Mar. 2009. IEEE Computer Society. 55

[23] J. Carter, S. Ozev, and D. Sorin. Circuit-level modeling for concurrent testing of operational defects due to gate oxide breadkown. In *Proc. of the 2005 Design, Automation and Test in Europe*, pages 300–305, June 2005. 13, 14, 47

[24] J. Choi, C. Cher, , H. Franke, H. Haman, A. Wedger, and P. Bose. Thermal-aware task scheduling at the system software level. In *Proc. of the 2007 International Symposium on Low Power Electronics and Design*, pages 213–218, Aug. 2007. 43

[25] P. C. Chu and J. E. Beasley. A genetic algorithm for the generalised assignment problem. *Microelectronic Engineering*, 24(1):17–23, 1997. 61

[26] M. de Kruijf, S. Nomura, and K. Sankaralingam. Relax: An architectural framework for software recovery of hardware faults. In *Proc. of the 37th Annual International Symposium on Computer Architecture*, pages 497–508, June 2010. 123, 141, 147, 148

[27] J. Donald and M. Martonosi. Techniques for multicore thermal management: Classification and new exploration. In *Proc. of the 33rd Annual International Symposium on Computer Architecture*, June 2006. 43

[28] X. Dong, Y. Xie, N. Muralimanohar, and N. P. Jouppi. A case study of incremental and background hybrid in-memory checkpointing. In *Proc. of the 2010 Exascale Evaluation and Research Techniques Workshop*, 2010. 115, 119, 147

[29] J. Duell, P. Hargrove, and E. Roman. The design and implementaion of berkeley lab's linux checkpoint/restart, 2002. 119

[30] D. Ernst, S. Das, S. Lee, D. Blaauw, T. Austin, T. Mudge, N. S. Kim, and K. Flautner. Razor: Circuit-level correction of timing errors for low-power operation. In *Proc. of the 37th Annual International Symposium on Microarchitecture*, pages 10–20, 2004. 43

[31] D. Ernst, N. S. Kim, S. Das, S. Pant, T. Pham, R. Rao, C. Ziesler, D. Blaauw, T. Austin, and T. Mudge. Razor: A low-power pipeline based on circuit-level timing speculation. In *Proc. of the 36th Annual International Symposium on Microarchitecture*, pages 7–18, 2003. 12

[32] R. Ernst and W. Ye. Embedded program timing analysis based on path clustering and architecture classification. pages 598–604, 1997. 128

[33] C. Evangs-Pughe. Live fast, die young [nanometer-scale ic life expectancy]. *IEE Review*, 50(7):34–37, 2004. 65

[34] S. Feng, S. Gupta, A. Ansari, and S. Mahlke. Shoestring: Probabilistic soft-error reliability on the cheap. In *18th International Conference on Architectural Support for Programming Languages and Operating Systems*, Mar. 2010. 115, 118, 132, 139, 145, 146

[35] S. Feng, S. Gupta, and S. Mahlke. Olay: Combat the signs of aging with intropsective reliability management. In *Proc. of the Workshop on Architectural Reliability*, June 2008. 64

[36] J. Friedrich et al. Desing of the power6 microprocessor, Feb. 2007. In *Proc. of ISSCC*. 56

[37] M. Gomaa and T. Vijaykumar. Opportunistic transient-fault detection. In *Proc. of the 32nd Annual International Symposium on Computer Architecture*, pages 172–183, June 2005. 7, 11, 40, 110

[38] M. A. Gomaa, C. Scarbrough, I. Pomeranz, and T. N. Vijaykumar. Transient-fault recovery for chip multiprocessors. In *Proc. of the 30th Annual International Symposium on Computer Architecture*, pages 98–109, 2003. 2, 7, 76

[39] S. Gupta, S. Feng, A. Ansari, J. A. Blome, and S. Mahlke. The stagenet fabric for constructing resilient multicore systems. In *Proc. of the 41st Annual International Symposium on Microarchitecture*, pages 141–151, 2008. 54

[40] C. Isci, G. Contreras, and M. Martonosi. Live, runtime phase monitoring and prediction on real systems with application to dynamic power management. In *Proc. of the 39th Annual International Symposium on Microarchitecture*, pages 359–370, Dec. 2006. 56

[41] ITRS. International technology roadmap for semiconductors 2008, 2008. http://www.itrs.net/. 10

[42] K. Kang, K. Kim, A. E. Islam, M. A. Alam, and K. Roy. Characterization and estimation of circuit reliability degradation under nbti using on-line iddq measurement. In *Proc. of the 44th Design Automation Conference*, June 2007. 55

[43] T. Kehl. Hardware self-tuning and circuit performance monitoring. In *Proc. of the 1993 International Conference on Computer Design*, pages 188–192, Oct. 1993. 40

[44] J. Kim, N. Hardavellas, K. Mai, B. Falsafi, and J. C. Hoe. Multi-bit Error Tolerant Caches Using Two-Dimensional Error Coding. In *Proc. of the 40th Annual International Symposium on Microarchitecture*, 2007. 83

[45] S. W. Kim, C.-L. Ooi, R. Eigenmann, B. Falsafi, and T. Vijaykumar. Reference idempotency analysis: A framework for optimizing speculative execution. In *Proc. of the 6th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 2–11, 2001. 123

[46] E. Kursun and C.-Y. Cher. Variation-aware thermal characterization and management of multi-core architectures. In *Proc. of the 2008 International Conference on Computer Design*, pages 280–285, Oct. 2008. 48

[47] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proc. of the 2004 International Symposium on Code Generation and Optimization*, pages 75–86, 2004. 101

[48] M.-L. Li, P. Ramachandran, S. K. Sahoo, S. V. Adve, V. S. Adve, and Y. Zhou. Understanding the propagation of hard errors to software and implications for resilient system design. In *16th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 265–276, 2008. 77, 87, 100, 112, 115

[49] X. Li, B. Huang, J. Qin, X. Zhang, M. Talmor, Z. Gur, and J. B. Bernstein. Deep submicron cmos integrated circuit reliability simulation with spice. In *Proc. of the 2005 International Symposium on Quality of Electronic Design*, pages 382–389, Mar. 2005. 53

[50] X. Li and D. Yeung. Application-level correctness and its impact on fault tolerance. In *Proc. of the 13th International Symposium on High-Performance Computer Architecture*, pages 181–192, Feb. 2007. 89, 147

[51] B. P. Linder and J. H. Stathis. Statistics of progressive breakdown in ultra-thin oxides. *Microelectronic Engineering*, 72(1-4):24–28, 2004. 13, 15, 24, 25, 47

[52] Z. Lu, J. Lach, M. R. Stan, and K. Skadron. Improved thermal management with reliability banking. *IEEE Micro*, 25(6):40–49, Nov. 2005. 43, 48, 49

[53] T. May and M. Woods. Alpha-particle-induced soft errors in dynamic memories. *IEEE Transactions on Electron Devices*, 26(1):2–9, Jan. 1979. 75

[54] S. McCamant and M. D. Ernst. Quantitative information flow as network flow capacity. In *Proc. of the SIGPLAN '08 Conference on Programming Language Design and Implementation*, pages 193–205, June 2008. 89

[55] A. Meixner, M. Bauer, and D. Sorin. Argus: Low-cost, comprehensive error detection in simple cores. *IEEE Micro*, 28(1):52–59, 2008. 2, 8, 110, 146

[56] S. Mishra and M. P. adn Douglas L. Goodman. In-situ sensors for product reliability monitoring, 2006. http://www.ridgetop-group.com/. 12

[57] F. Monsieur, E. Vincent, D. Roy, S. Bruyere, J. C. Vildeuil, G. Pananakakis, and G. Ghibaudo. A thorough investigation of progressive breakdown in ultra-thin oxides. physical understanding and application for industrial reliability assessment. In *Proc. of the 2002 International Reliability Physics Symposium*, pages 45–54, Apr. 2002. 15

[58] P. Montesinos, W. Liu, and J. Torrellas. Using register lifetime predictions to protect register files against soft errors. In *Proc. of the 2007 International Conference on Dependable Systems and Networks*, pages 286–296, 2007. 112

[59] S. S. Mukherjee, M. Kontz, and S. K. Reinhardt. Detailed design and evaluation of redundant multithreading alternatives. In *Proc. of the 29th Annual International Symposium on Computer Architecture*, pages 99–110, 2002. 2, 76, 115, 146

[60] S. S. Mukherjee, C. Weaver, J. Emer, S. Reinhardt, and T. Austin. A systematic methodology to compute the architectural vulnerability factors for a high performance microprocessor. In *International Symposium on Microarchitecture*, pages 29–42, Dec. 2003. 111

[61] H. G. Naik, R. Gupta, and P. Beckman. Analyzing checkpointing trends for applications on the ibm blue gene/p system. *Proc. of the 2009 International Conference on Parallel ProcessingWorkshops*, pages 81–88, 2009. 119

[62] N. Nakka, K. Pattabiraman, and R. Iyer. Processor-level selective replication. In *Proc. of the 2007 International Conference on Dependable Systems and Networks*, 2007. 146

[63] J. V. Neumann. Probabilistic logics and the synthesis of reliable organisms from unreliable components. *Automata Studies*, pages 43–98, 1956. 6

[64] N. Oh, S. Mitra, and E. J. McCluskey. Ed4i: Error detection by diverse data and duplicated instructions. *IEEE Transactions on Computers*, 51(2):180–199, 2002. 94

[65] OpenCores. OpenRISC 1200, 2006. http://www.opencores.org/projects.cgi/web/or1k/openrisc_1200. 21

[66] K. Pattabiraman, Z. Kalbarczyk, and R. K. Iyer. Automated derivation of application-aware error detectors using static analysis: The trusted illiac approach. *IEEE Transactions on Dependable and Secure Computing*, 99(PrePrints), 2009. 146

[67] M. Powell, M. Gomaa, and T. Vijaykumar. Heat-and-run: Leveraging smt and cmp to manage power density through the operating system. In *12th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 260–270, Oct. 2004. 43, 48

[68] M. Prvulovic, Z. Zhang, and J. Torrellas. Revive: cost-effective architectural support for rollback recovery in shared-memory multiprocessors. *Proc. of the 29th Annual International Symposium on Computer Architecture*, pages 111–122, 2002. 119, 147

[69] P. Racunas, K. Constantinides, S. Manne, and S. Mukherjee. Perturbation-based fault screening. In *Proc. of the 13th International Symposium on High-Performance Computer Architecture*, pages 169–180, Feb. 2007. 112, 146

[70] J. Ray, J. Hoe, and B. Falsafi. Dual use of superscalar datapath for transient-fault detection and recovery. In *Proc. of the 34th Annual International Symposium on Microarchitecture*, pages 214–224, Dec. 2001. 7, 11, 40

[71] V. Reddy, S. Parthasarathy, and E. Rotenberg. Understanding prediction-based partial redundant threading for low-overhead, high-coverage fault tolerance. In *14th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 83–94, Oct. 2006. 11, 40, 110, 146

[72] V. Reddy and E. Rotenberg. Inherent time redundancy (itr): Using program repetition for low-overhead fault tolerance. In *Proc. of the 2007 International Conference on Dependable Systems and Networks*, pages 307–316, June 2007. 110

[73] V. K. Reddy and E. Rotenberg. Coverage of a microarchitecture-level fault check regimen in a superscalar processor. In *Proc. of the 2008 International Conference on Dependable Systems and Networks*, pages 1–10, June 2008. 110

[74] S. K. Reinhardt and S. S. Mukherjee. Transient fault detection via simulataneous multithreading. In *Proc. of the 27th Annual International Symposium on Computer Architecture*, pages 25–36, June 2000. 2, 11, 76, 83, 110, 115

[75] G. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August. SWIFT: Software implemented fault tolerance. In *Proc. of the 2005 International Symposium on Code Generation and Optimization*, pages 243–254, 2005. 8, 11, 77, 83, 101, 107, 111, 146

[76] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, D. I. August, and S. S. Mukherjee. Software-controlled fault tolerance. *ACM Transactions on Architecture and Code Optimization*, 2(4):366–396, 2005. 77, 94, 111

[77] D. Roberts, R. Dreslinski, E. Karl, T. Mudge, D. Sylvester, and D. Blaauw. When homogeneous becomes heterogeneous: Wearout aware task scheduling for streaming applications. In *Proc. of the Workshop on Operationg System Support for Heterogeneous Multicore Architectures*, Sept. 2007. 42

[78] R. Rodriguez, J. H. Stathis, and B. P. Linder. Modeling and experimental verification of the effect of gate oxide breakdown on cmos inverters. In *International Reliability Physics Symposium*, pages 11–16, Apr. 2003. 15, 16

[79] B. F. Romanescu and D. J. Sorin. Core cannibalization architecture: Improving lifetime chip performance for multicore processor in the presence of hard faults. In *Proc. of the 17th International Conference on Parallel Architectures and Compilation Techniques*, 2008. 54

[80] E. Rotenberg. AR-SMT: A microarchitectural approach to fault tolerance in microprocessors. In *International Symposium on Fault Tolerant Computing*, pages 84–91, 1999. 2, 7, 11, 40, 76, 110, 115, 146

[81] S. K. Sahoo, M.-L. Li, P. Ramchandran, S. Adve, V. Adve, and Y. Zhou. Using likely program invariants for hardware reliability. In *Proc. of the 2008 International Conference on Dependable Systems and Networks*, 2008. 146

[82] S. Sarangi, B. Greskamp, R. Teodorescu, J. Nakano, A. Tiwari, and J. Torrellas. Varius: A model of process variation and resulting timing errors for microarchitects. In *IEEE Transactions on Semiconductor Manufacturing*, pages 3–13, Feb. 2008. 64

[83] N. Shanbhag, R. Abdallah, R. Kumar, and D. Jones. Stochastic computation. In *Proc. of the 47th Design Automation Conference*, 2010. 147

[84] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *Tenth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 45–57, New York, NY, USA, 2002. ACM. 98

[85] J. Shin, V. V. Zyuban, P. Bose, and T. M. Pinkston. A proactive wearout recovery approach for exploiting microarchitectural redundancy to extend cache sram lifetime. In *Proc. of the 35th Annual International Symposium on Computer Architecture*, pages 353–362, 2008. 8

[86] P. Shivakumar, S. Keckler, C. Moore, and D. Burger. Exploiting microarchitectural redundancy for defect tolerance. In *Proc. of the 2003 International Conference on Computer Design*, page 481, Oct. 2003. 40

[87] P. Shivakumar, M. Kistler, S. Keckler, D. Burger, and L. Alvisi. Modeling the effect of technology trends on the soft error rate of combinational logic. In *Proc. of the 2002 International Conference on Dependable Systems and Networks*, pages 389–398, June 2002. 79

[88] K. Skadron, M. R. Stan, K. Sankaranarayanan, W. Huang, S. Velusamy, and D. Tarjan. Temperature-aware microarchitecture: Modeling and implementation. *ACM Transactions on Architecture and Code Optimization*, 1(1):94–125, 2004. 10, 23, 48, 64

[89] J. Sloan, D. Kesler, R. Kumar, and A. Rahimi. A numerical optimization-based methodology for application robustification: Transforming applications for error tolerance. In *Proc. of the 2010 International Conference on Dependable Systems and Networks*, 2010. 147

[90] J. Smolens, J. Kim, J. Hoe, and B. Falsafi. Efficient resource sharing in concurrent error detecting superscalar microarchitectures. In *Proc. of the 37th Annual International Symposium on Microarchitecture*, pages 256–268, Dec. 2004. 2, 7, 11, 40, 76

[91] J. C. Smolens, B. T. Gold, B. Falsafi, and J. C. Hoe. Reunion: Complexity-effective multicore redundancy. In *Proc. of the 39th Annual International Symposium on Microarchitecture*, pages 223–234, 2006. 7, 111, 146

[92] A. Sodani and G. Sohi. Dynamic instruction reuse. In *Proc. of the 25th Annual International Symposium on Computer Architecture*, pages 194–205, June 1998. 40

[93] D. Sorin. *Fault Tolerant Computer Architecture*. Morgan and Claypool, San Rafael, CA, 2009. 9

[94] D. Sorin, M. Martin, M. Hill, and D. Wood. Safetynet: improving the availability of shared memory multiprocessors with global checkpoint/recovery. *Proc. of the 29th*

Annual International Symposium on Computer Architecture, pages 123–134, 2002. 119, 147

[95] L. Spainhower and T. Gregg. IBM S/390 Parallel Enterprise Server G5 Fault Tolerance: A Historical Perspective. *IBM Journal of Research and Development*, 43(6):863–873, 1999. 2, 7, 11, 76, 115, 119

[96] V. Sridharan and D. R. Kaeli. Eliminating microarchitectural dependency from architectural vulnerability. In *Proc. of the 15th International Symposium on High-Performance Computer Architecture*, pages 117–128, 2009. 147, 148

[97] J. Srinivasan, S. V. Adve, P. Bose, and J. A. Rivers. The case for lifetime reliability-aware microprocessors. In *Proc. of the 31st Annual International Symposium on Computer Architecture*, pages 276–287, June 2004. 22, 53

[98] J. Srinivasan, S. V. Adve, P. Bose, and J. A. Rivers. Exploiting structural duplication for lifetime reliability enhancement. In *Proc. of the 32nd Annual International Symposium on Computer Architecture*, pages 520–531, June 2005. 12, 40

[99] StockCharts.com. *TRIX*, Oct. 2006. http://stockcharts.com/education/IndicatorAnalysis/indic_trix.h 29

[100] J. Sune and E. Wu. From oxide breakdown to device failure: an overview of post-breakdown phenomena in ultrathin gate oxides. In *International Conference on Integrated Circuit Design and Technology*, pages 1–6, May 2006. 15

[101] D. Sylvester, D. Blaauw, and E. Karl. Elastic: An adaptive self-healing architecture for unpredictable silicon. *IEEE Journal of Design and Test*, 23(6):484–490, 2006. 44, 54

[102] J.-J. Tang, K.-J. Lee, and B.-D. Liu. A practical current sensing technique for iddq testing. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 3(2):302–310, June 1995. 55

[103] R. Teodorescu and J. Torrellas. Variation-aware application scheduling and power management for chip multiprocessors. In *Proc. of the 35th Annual International Symposium on Computer Architecture*, pages 363–374, June 2008. 42, 43, 48

[104] A. Tiwari, S. Sarangi, and J. Torrellas. Recycle: Pipeline adaptation to tolerate process variation. In *Proc. of the 34th Annual International Symposium on Computer Architecture*, pages 323–334, June 2007. 43

[105] A. Tiwari and J. Torrellas. Facelift: Hiding and slowing down aging in multicores. In *Proc. of the 41st Annual International Symposium on Microarchitecture*, pages 129–140, Dec. 2008. 8, 42, 48, 50

[106] N. Vachharajani, M. J. Bridges, J. Chang, R. Rangan, G. Ottoni, J. A. Blome, G. A. Rei, M. Vachharajani, and D. I. August. Rifle: An architectural framework for user-centric information-flow security. In *Proc. of the 37th Annual International Symposium on Microarchitecture*, pages 243–254, Dec. 2004. 89

[107] T. Vijaykumar, I. Pomeranz, and K. Cheng. Transient-fault recovery via simultaneous multithreading. In *Proc. of the 29th Annual International Symposium on Computer Architecture*, pages 87–98, May 2002. 7, 11, 40

[108] C. Wang, H. seop Kim, Y. Wu, and V. Ying. Compiler-managed software-based redundant multi-threading for transient fault detection. In *Proc. of the 2007 International Symposium on Code Generation and Optimization*, 2007. 110, 146

[109] N. Wang and S. Patel. Restore: Symptom based soft error detection in microprocessors. In *International Conference on Dependable Systems and Networks*, pages 30–39, June 2005. 80, 111

[110] N. J. Wang, M. Fertig, and S. J. Patel. Y-branches: When you come to a fork in the road, take it. In *Proc. of the 12th International Conference on Parallel Architectures and Compilation Techniques*, pages 56–65, 2003. 84, 94

[111] N. J. Wang and S. J. Patel. ReStore: Symptom-based soft error detection in microprocessors. *IEEE Transactions on Dependable and Secure Computing*, 3(3):188–201, June 2006. 77, 82, 87, 88, 98, 100, 111, 115, 118, 132, 145

[112] N. J. Wang, J. Quek, T. M. Rafacz, and S. J. Patel. Characterizing the Effects of Transient Faults on a High-Performance Processor Pipeline. In *International Conference on Dependable Systems and Networks*, page 61, June 2004. 97

[113] N. J. Warter and W.-M. W. Hwu. A software based approach to achieving optimal performance for signature control flow checking. In *Proc. of the 1998 International Symposium on Fault-Tolerant Computing*, pages 442–449, 1990. 120, 146

[114] C. Weaver and T. M. Austin. A fault tolerant approach to microprocessor design. In *Proc. of the 2001 International Conference on Dependable Systems and Networks*, pages 411–420, Washington, DC, USA, 2001. IEEE Computer Society. 2, 76

[115] P. M. Wells, K. Chakraborty, and G. S. Sohi. Mixed-mode multicore reliability. In *17th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 169–180, 2009. 111

[116] J. Winter and D. Albonesi. Scheduling algorithms for unpredictably heterogeneous cmp architectures. In *Proc. of the 2008 International Conference on Dependable Systems and Networks*, pages 42–51, June 2008. 42, 43, 48, 56

[117] E. Wu, J. M. McKenna, W. Lai, E. Nowak, and A. Vayshenker. Interplay of voltage and temperature acceleration of oxide breakdown for ultra-thin gate oxides. *Solid-State Electronics*, 46:1787–1798, 2002. 22

[118] T. J. Yamaguchi, M. Soma, D. Halter, J. Nissen, R. Raina, M. Ishida, and T. Watanabe. Jitter measurements of a powerpc microprocessor using an analytic signal method. In *Proc. of the 2000 International Test Conference*, pages 955–964, 2000. 38

[119] X. Yang, E. Weglarz, and K. Saluja. On nbti degradation process in digital logic circuits. In *Proc. of the 2007 International Conference on VLSI Design*, pages 723–730, Jan. 2007. 13, 14, 47

[120] M. T. Yourst. Ptlsim: A cycle accurate full system x86-64 microarchitectural simulator. In *Proc. of the 2007 IEEE Symposium on Performance Analysis of Systems and Software*, pages 23–34, 2007. 97

[121] J. F. Ziegler and H. Puchner. *SER-History, Trends, and Challenges: A Guide for Designing with Memory ICs*. Cypress Semiconductor Corp., 2004. 76