CARNEGIE MELLON UNIVERSITY

CARNEGIE INSTITUTE OF TECHNOLOGY
DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

Fingerprinting: Hash-Based Error Detection in Microprocessors

Jared C. Smolens

January 2008

Abstract

Today's commodity processors are tuned primarily for performance and power. As CMOS scaling continues into the deep sub-micron regime, soft errors and device wearout will increasingly jeopardize the reliability of unprotected processor pipelines. To preserve reliable operation, processor cores will require mechanisms to detect errors affecting the control and datapaths. Conventional techniques such as parity, error correcting codes, and self-checking circuits have high implementation overheads and therefore these techniques cannot be easily applied to complex and timing-critical high-performance pipelines.

This thesis proposes and evaluates architectural and microarchitectural *fingerprints*. A fingerprint is a compact (e.g., 16-bit) signature of recent architectural or microarchitectural state updates. By periodically comparing a fingerprint and a reference over an interval of execution, the system can detect errors in a timely and bandwidth-efficient manner. Architectural fingerprints capture in-order architectural state with lightweight monitoring hardware at the retirement stages of a pipeline, while microarchitectural fingerprints leverage existing design-for-test hardware to accumulate a signature of internal state.

This thesis explores two applications of fingerprints. In the Reunion execution model, this thesis shows that architectural fingerprints can detect both soft errors and input incoherence with complexity-effective redundant execution in a chip multiprocessor. Cycle-accurate simulation shows that the performance overhead is only 5-6% over more complicated designs that strictly replicate inputs. In another application, FIRST, fingerprints detect emerging wearout faults by periodically testing the processor under marginal operating conditions. Wearout fault simulation in a commercial processor show that architectural fingerprints have high coverage of widespread wearout, while microarchitectural fingerprints provide superior coverage of both individual and widespread wearout.

Acknowledgements

First, I thank Professor James C. Hoe, my academic advisor, for guiding and supporting me in my graduate studies and providing a comfortable and productive environment in which to do research. His comments, criticism, and insight over the years have greatly refined my skills in writing, speaking, and independent research.

I also thank Professor Babak Falsafi for his guidance and advice. He has constantly provided me with motivation and a valuable second opinion throughout my time in the graduate program. I also thank my committee members Professor Shawn Blanton and Dr. Shubu Mukherjee for their time, patience, and helpful comments. Their feedback has substantially improved this thesis.

I thank my mentor at Intel Corporation, TM Mak, for supplying me with abundant information on current processor designs and using his extensive contacts to gain access to commercial designs for my research. Without his help and Intel's cooperation, there would be many more open, or even unrecognized, questions in this work. I also thank Sun Microsystems for their generous and timely release of the OpenSPARC RTL and architecture tools. With these resources I was able to quickly prototype and model many of the key ideas in this thesis. Finally, I thank the organizations that funded my research, including the National Science Foundation, Intel Corporation, and the Center for Circuit and System Solutions (C2S2).

My fellow graduate students in the TRUSS group and CALCM also deserve abundant thanks for all of the ways in which they have helped me over the years, including, but not limited to, writing and debugging simulators, coauthoring papers, proofreading papers and this thesis, fixing my talks, and tolerating my awful puns. In order of arrival to the graduate program, I specifically want to acknowledge Tom Wenisch, Roland Wunderlich, Jangwoo Kim, Stephen Somogyi, Nikos Hardavellas, Brian Gold, Eric Chung, and Eriko Nurvitadhi. I also thank Professor Charles Neuman for his advice and running commentary on the state of the world.

I am grateful for the long-standing encouragement, advice, and conversation from my friends and co-workers at Unisys Corporation, particularly Emilio Salgueiro and John Black. My experiences with the large computer systems at Unisys gave me a unique perspective on the industry and cemented my interest and curiosity for computer architecture.

I am thankful for the endless patience, love, and support I have received from my family, Gene, Susan, and Max, and from Yang Wang, who still stays close even when far away.

Finally, I must recognize those who plead: *just make it perfect, how hard can it be?*

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Reliable computer systems form the backbone of critical information technology infrastructure in today's society. Once billed as the reliable substrate for microprocessors, CMOS circuits are now widely expected to suffer from increasing levels of soft errors and wearout phenomena. With process technologies entering the deep sub-micron regime, process engineers predict that per-chip radiation-induced single event upsets in latches and unprotected SRAM will increase exponentially [48, 96]. New sources of intermittent faults will emerge, including process variation, narrowing voltage and thermal noise margins, and aggressive guardbands [22]. Processors will also experience "lifetime reliability" effects, where the smaller device dimensions mean that individual transistors and wires become increasingly susceptible to wearout phenomena including gate oxide breakdown, negative-bias temperature instability, hot-carrier injection, and electromigration [65, 110].

This thesis investigates mechanisms for detecting errors that occur at runtime in a processor core, focusing on two growing threats in modern processors: radiation-induced soft errors and device wearout.

Soft Errors. A soft error results from a transient bit upset in a digital circuit. These upsets can occur from sources such as neutrons and alpha particles striking the silicon [121]. Because these upsets originate from external physical sources, they occur at a constant rate over a processor's lifetime; however they cause no permanent damage to the underlying circuit and are therefore considered "soft." When the architectural state (e.g., architectural registers and store values and addresses) differs from the specified behavior, an architectural error is said to result. An undetected architectural error, called silent data corruption, is a serious concern because these errors can result in incorrect program execution or unexpected application crashes. Architectural errors that are detected, but uncorrected, decrease the reliability and availability of the system; however, the user can be alerted to the possibility of data corruption. With timely error detection mechanisms, a computer system can recover and maintain correct execution with rollback to a checkpoint or voting across redundant units.

Device Wearout. Device wearout presents another challenge to reliability. These faults develop initially as "soft breakdown" events that cause transistor switching speeds to gradually slow, while the underlying logical functions are preserved [60, 89]. Therefore, these faults initially do not produce architectural errors during normal execution conditions, but can still be observed by removing the voltage and frequency guardbands. Wearout faults steadily worsen with continued operation. If emerging wearout faults can be revealed in a test before being exposed in normal operating conditions, the system can be repaired or replaced before general-purpose execution is affected, thus preserving the system's reliability and correct operation. Eventually, the wearout faults develop into "hard breakdown," where a device fails catastrophically. Detecting these permanent failures is out of the scope of this thesis.

Without mechanisms to detect soft errors and device wearout, maintaining processor reliability will be an increasing hardship for system designers. Today's processor pipelines are largely unprotected and existing solutions for protection impose unacceptable performance and area overheads or are only applicable to specific functional units.

1.1 Problem and Scope

This thesis addresses the problem of timely error detection inside the processor pipeline. Figure 1 illustrates the problem. The system under test—a processor with faults in its datapath—is compared with a reference execution. The error detection mechanism is responsible for continuously comparing the two executions and detecting differences caused by the underlying faults at runtime. Over an interval of execution, the error detection mechanism must determine whether an

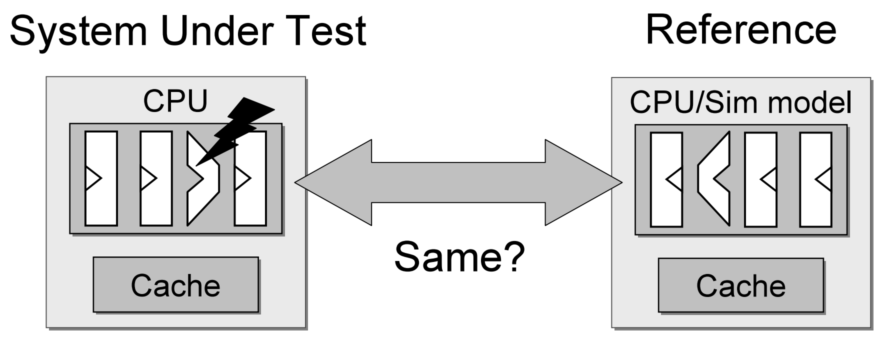


Figure 1. General error detection between a system under test and a reference.

error affects execution during the interval or the execution is free of errors. The reference execution can take many forms that mirror the system under test's behavior, including another processor or execution context in an symmetric or asymmetric configuration, a register-transfer level (RTL) model, or even a functional instruction set simulator.

This thesis presents two instantiations of the problem. (1) A concurrent error detection and recovery microarchitecture, called Reunion, where dual-modular redundant (DMR) processor cores symmetrically compact and compare their results against each other to detect soft errors. (2) An in-field wearout detection procedure, called FIRST, where the processor execution is compacted and compared against itself or fault-free RTL models with reduced guardbands to detect developing wearout faults.

Error detection in this thesis is limited to the datapath and control logic in unretired state contained in the processor core. Regular array structures such as caches and architectural register files have well-known and effective information redundancy mechanisms, including parity and error correcting codes (ECC) [100], which are complementary to this work. The complex design and timing-critical nature of high-performance processors precludes the use of such techniques throughout the pipeline.

This thesis identifies three metrics for evaluating error detection mechanisms:

1. *Detection latency* indicates the length of time from a fault's initial occurrence to its detection as an error. This measure is important for ensuring that entire checkpoints and intervals of execution results are free of errors.

2. *Comparison bandwidth* is the amount of state that must be compared to check that an execution interval is free of errors. Because buses and pins are a limited resource, the mechanism's bandwidth requirements for periodic and timely are critical to an implementation's feasibility.

3. *Error coverage* is the probability of detecting a given error in a timely fashion to successfully isolate or correct the error. Error coverage must be high enough to meet the desired system reliability goals, but is rarely perfect in practice [71].

The scope of this thesis is further constrained to error detection mechanisms that tolerate non-determinism and function at-speed and at runtime (in the customer's environment). Guaranteeing deterministic behavior during runtime is impractical in many commercial designs. Detection mechanisms must allow some non-deterministic behavior while minimizing false positives and negatives. Furthermore, detection mechanisms must function properly at-speed (i.e., match the core frequency). This requirement is necessary because for soft error detection, processors are only useful products if they run at full speed, while for wearout the fault's initial onset is obscured at lower frequencies. Finally, these detection mechanisms are only useful if they can function at runtime in a customer's environment.

1.2 Fingerprints

This thesis proposes and evaluates a hash-based detection mechanism, called *Fingerprinting*. A fingerprint is a compact signature (e.g., 16 bits) of a processor's updates to architectural or microarchitectural state which is periodically compared with a corresponding signature from a reference to detect errors. Fingerprinting is conceptually illustrated in Figure 2. Fingerprinting addresses the three metrics above, while permitting limited non-determinism and at-speed detection. They address detection latency by moving the point of detection close to the actual fault. Fingerprints bound the detection bandwidth by compacting updated state into a signature that concisely represents the sequence of state updates produced by the processor over an interval of execution. Finally, by carefully constructing the signature to avoid aliasing, the fingerprint can be engineered to have enough coverage to meet the design's reliability budget [71].

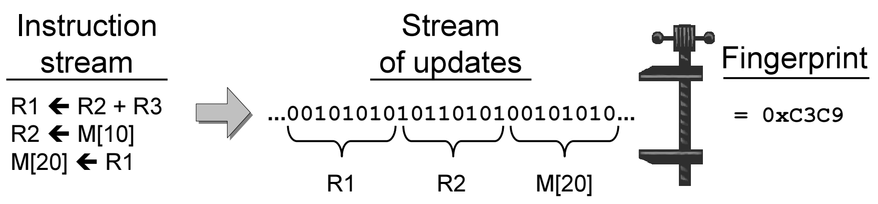


Figure 2: Fingerprints compress architectural and microarchitectural state updates into a compact hash.

1.3 Thesis contributions

This thesis proposes and evaluates two realizations of fingerprinting:

- **Architectural fingerprints.** Architectural fingerprints calculate a deterministic hash of architectural state updates in processors. They permit timely detection of errors that propagate to architectural state. This thesis presents a design and implementation of architectural fingerprints in commercial processor RTL. The results show that an architectural fingerprint unit in a proof-of-concept redundant execution RTL prototype adds less than 4% in area to an already very simple pipeline. Statistical fault injection experiments concretely demonstrate that architectural fingerprints are effective for detecting both soft errors state and widespread device wearout.

- **Microarchitectural fingerprints.** Microarchitectural fingerprints calculate a deterministic hash of microarchitectural state updates internal to a processor. These fingerprints allow spatial and temporal localization of faults within a processor, including those not propagated to architectural state, using existing design-for-test hardware with slight modifications to avoid common sources of non-determinism. This thesis proposes a design for microarchitectural fingerprints in a commercial processor.

This thesis makes the following contributions from studying the feasibility, implementation, and applications of fingerprints:

- **Scalable Hash Architecture.** This thesis proposes and evaluates a scalable hash architecture for accumulating architectural fingerprints. The proposed hash architecture, an X-compact-like [69] spatial and MISR-based [93] temporal compactor—based on traditional manufacturing test compaction architectures—is modified to compact retiring architectural state. The design preserves evidence of errors in the output hash as effectively as an ideal cyclic redundancy check (CRC), but can scale to accept the output from wide-issue superscalar processors at a fraction of an equivalent CRC implementation's area and latency.

- **Reunion.** Reunion is a formal execution model for redundant execution across loosely-coupled redundant cores in a chip multiprocessor (CMP), using architectural fingerprints. This thesis shows that a key problem in redundant execution, called input incoherence, can be detected and handled with the same mechanisms needed for soft error detection and recovery. The evaluation demonstrates that complexity-effective redundant execution for shared-memory programs in a CMP can be achieved with only a 5-6% performance overhead over more complicated solutions that use strict input replication.

- **Fingerprinting in Reliability and Self Test (FIRST).** FIRST is a procedure for in-field wearout detection using microarchitectural fingerprints. FIRST identifies emerging wearout faults before they affect general execution. The study demonstrates that microarchitectural fingerprints are effective for observing both individual and widespread wearout faults. Furthermore, the study shows that architectural fingerprints are equivalent to microarchitectural fingerprints for detecting widespread wearout faults.

The remainder of this thesis is organized as follows. Chapter 2 introduces background and the concept of architectural fingerprints. Chapter 3 explores the implementation of architectural fingerprints in two modern commercial microarchitectures. Chapter 4 explores a range of hash designs for architectural fingerprints. Chapter 5 presents and evaluates the Reunion execution model, an application of architectural fingerprints. Microarchitectural fingerprints are introduced in Chapter 6. Chapter 7 presents and evaluates FIRST for both architectural and microarchitectural fingerprints. Related work is discussed in Chapter 8. This thesis concludes and identifies future research directions in Chapter 9.

Chapter 2

Architectural Fingerprints

Architectural fingerprints summarize into a compact signature the in-order architectural state updates—the general-purpose register writes and stores to memory—generated by a processor core. Architectural fingerprints expose errors in architectural state that otherwise have a long error detection latency. By periodically comparing small signatures that summarize the monitored state, entire intervals of execution are compared instantly and the comparison bandwidth can be reduced by orders of magnitude over directly comparing each architectural state update. Finally, by leveraging strong but efficient hash functions, fingerprints can avoid aliasing and therefore maintain high error coverage.

This chapter is organized as follows. The important terminology and fault model for this thesis are introduced in Section 2.1. The architectural fingerprint concept is defined in Section 2.2. The metrics for evaluating architectural fingerprints follow in Section 2.3. The following two chapters provide a study of implementing architectural fingerprints in commercial microarchitectures and a detailed study of the architectural hash design, respectively.

2.1 Fault model

This section introduces the terminology and fault model for the remainder of this thesis. The terminology in this thesis is adopted from the "minimum consensus" view in the dependable computing community [12].

An *architectural error* is a deviation from the defined correct architectural execution of a processor. In the context of architectural fingerprints, this term is abbreviated to *error*. This thesis is scoped to detecting errors that arise within the microprocessor core data and control paths. Unlike cache memories and external buses, which are excluded, these units are complex and irregular yet still performance-critical. Thus, the core data and control paths cannot easily be protected with traditional information redundancy mechanisms such as parity and ECC [100]. Errors propagating to SRAM arrays (which are increasingly protected by parity and ECC) can be detected as a side effect, but are not the focus of this work.

A *fault* is the hypothesized source of an error. The detection mechanisms considered in this thesis directly detect errors, not the underlying fault. Errors can propagate between components of the processor and produce. However, the error can also remain internal to the processor core or disappear altogether. In this case, the error is *masked*.

Masking. Masking occurs on many levels of abstraction. These are summarized below. Errors can be masked *electrically*, for example, an energetic partial strike can generate a transient glitch in logic. This glitch is attenuated by passing through several levels of combinational logic gates. *Logical* masking occurs when controlling inputs on a cone of combinational logic prevent a glitch from propagating. Glitches that reach a latch can still encounter *latch-window* masking unless they arrive in the time window when an input value is recorded by the latch [97]. Latched errors can still be *architecturally* masked if the latched error is overwritten before propagating to architectural registers or store values [73]. Finally, software can also overwrite or ignore errors propagated to architectural values through *program-level* masking. The effects of electrical, latch-window, and program-level masking are out of the scope of this thesis. Therefore, the architectural fingerprints presented in this thesis specifically contend with both logical and architectural masking.

The errors addressed by architectural fingerprints fall into two classes:

Soft Errors. Soft errors arise from transient faults that cause or more bit flips in a digital circuit. The underlying fault is a temporary upset and, unless captured in a sequential element such as a latch or SRAM, the effects completely disappear in a fraction of a clock cycle [17]. These faults include well-documented sources such as energetic particle strikes [121], and decreasing electrical and thermal noise margins [22].

Device Wearout. Errors from device wearout are intermittent or "elusive" faults which can be observed as one or more bit flips in a digital circuit. Wearout faults initially cause missed timing (e.g., setup times) in a correctly designed and manufactured circuit because the constituent logic gates and wires gradually operate more slowly. These faults include mechanisms such as the soft breakdown associated with gate oxide breakdown [60], negative-bias temperature instability [87], hot-carrier injection [28] and electromigration [43]. The onset of wearout is gradual over time and the fault activation is exacerbated by environmental and operating conditions such as increased temperature and frequency and reduced voltage [65].

This thesis is concerned with detecting wearout during soft breakdown. Detecting hard breakdown where the device ceases functioning entirely [60] is out of the scope of this thesis, although the presented techniques may also be effective for such faults.

2.2 Architectural Fingerprints

Architectural fingerprints are a compact, reproducible, deterministic hash of architectural state updates from a microprocessor. Architectural state consists of general-purpose registers, values stored in memory, and excludes non-architectural information such as cache misses, speculated execution, and timing. The architectural fingerprint is conceptually illustrated in Figure 2 of Chapter 1. Architectural fingerprints between two units—units that are supposed to execute the same program—are periodically compared to detect differences due to architectural errors in the redundant executions. The frequency of comparing architectural fingerprints, measured in instructions, is defined as the architectural fingerprint *comparison interval*. Architectural fingerprints summarize the entire set of architectural results over the comparison interval.

By selecting only architectural state, comparison of the two units is simplified. Rather than requiring redundant processors to execute a program cycle-for-cycle in precisely the same way, the executions need only generate the same results. For a processor core, this means architectural values are sampled for the fingerprint in program order from processor retirement buses, only on cycles when a value is being retired and not on cycles when the buses are idle. For other models, such as functional simulators, this comprises a complete program-order trace of execution. Because the hash is constructed from discrete instruction results, the architectural fingerprint is linked to precise architectural state on instruction boundaries. This property bounds the latency of detecting architectural errors to the fingerprint comparison interval. Furthermore, this eases integration with recovery mechanisms such as checkpoints [4, 66] and precise exception rollback [103], which are generally designed to operate on an architectural instruction granularity.

The redundant units compare fingerprints at regular instruction intervals and at points beyond which recovery mechanisms cannot cross (e.g., non-idempotent operations such as external I/O). Three possible outcomes can result from the comparison. First, the comparison can result in a fingerprint mismatch when the fingerprint values differ, signaling that at least one unit is in error. With two units, the fingerprint comparison can only identify differences, but the fingerprint alone cannot determine which unit is incorrect. Voting can disambiguate this situation with three or more redundant units. Second, the comparison can correctly signal a fingerprint match when no error is present. Finally, the comparison falsely signals a match when, in fact, an error is present but the hashes alias or a common-mode failure causes the same incorrect execution in all units. The final case signifies an undetected error and loss of error coverage.

2.3 Metrics

The overall goal of architectural fingerprinting is to detect an error in a processor's architectural state to prevent silent data corruption (SDC) or detected, uncorrectable errors (DUE) [73]. To achieve this, the error detection mechanism must balance three inter-related measures: detection latency, comparison bandwidth, and error coverage. Each of these requirements is discussed in detail below.

Detection Latency. The detection latency is the distance between an error occurring and its corresponding effects being observed by the detection mechanism. For architectural fingerprints, the detection latency is measured in instructions. The detection latency is governed by the granularity at which the redundant units are compared. Ideally, errors are detected precisely when and where

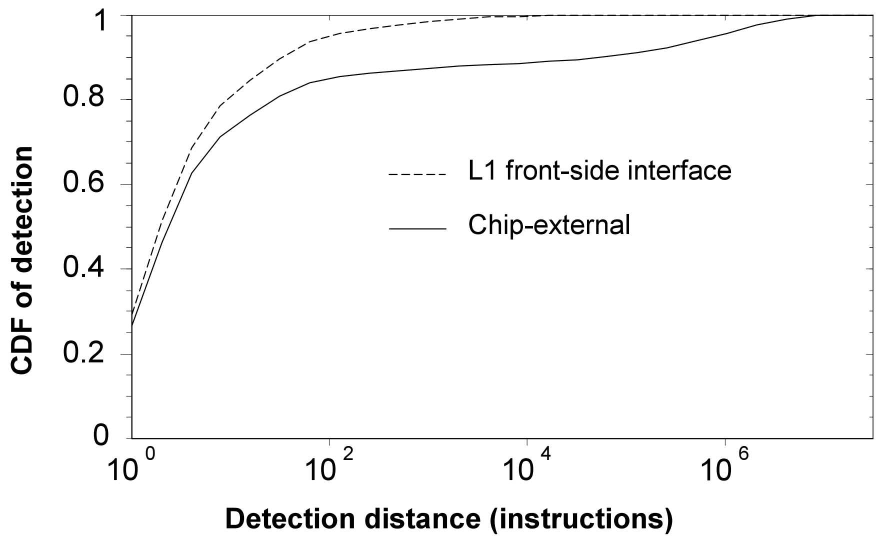


Figure 3: Cumulative distribution function (CDF) of errors detected as a function of instruction distance for a commercial OLTP workload.

they originate, however placing detection logic at every device in the processor is unrealistic. As the detection latency grows, isolating and recovering an error's effect becomes more difficult because the error can propagate to external components. Furthermore, rollback recovery becomes impossible if the processor has performed non-idempotent operations, such as network I/O, which once initiated, cannot be recalled. Hence, a hard bound on the detection latency is critical to maintaining reliability.

Traditional mainframes compare the outputs of lockstepped processors at the chip-external bus pins [16, 95]. These systems can detect errors in the values of writebacks of modified cache blocks or erroneous memory addresses that cause cache misses. While these systems are effective at detecting and isolating errors within a chip, they cannot guarantee the absence of latent errors in the processor core and caches (which have not yet propagated outside of the chip). Because of this long detection latency, recovery becomes complicated in these systems, requiring custom operating systems and applications to support rollback recovery using software checkpoints.

Several research proposals have also considered detection at the L1 cache write port [72, 88]. This observation location improves the detection latency over chip-external detection because errors must only be propagated to a store for detection. Architectural fingerprints can bring the detection latency down to individual instructions. At these short detection latencies,

The detection latency problem is now analyzed quantitatively. Figure 3 shows the fraction of errors in architectural state propagated to the chip-external and the L1 cache interface within a fixed number of instructions in an on-line transaction processing (OLTP) workload. OLTP is an important commercial workload and is representative of other workloads such as integer SPEC CPU benchmarks. The horizontal axis shows the distance, in instructions, until an error in an instruction result is observable at the L1 cache and chip-external interfaces in a system with a 1MB L2 cache, optimistically assuming program-level masking never occurs (program-level masking furthers increases the error detection latency for errors that are still eventually detected). The vertical axis shows the fraction of instructions with errors detected at that distance for each detection mechanism. The key observation is that a significant fraction of errors (10%) remains undetected, even after executing hundreds or thousands of instructions. These long—and potentially unbounded—latencies can make recovery impossible. Such situations include times when non-idempotent I/O operations (operations that cannot be repeated) have been retired in the meantime. Similarly, for recovery mechanisms within a processor core, such as precise exception rollback [85], error detection mechanisms at the L1 cache and further from the core are insufficient for recovery because errors can be irrevocably committed to architectural register state before they propagate to a store value. This result argues for comparing state updates directly, as with architectural fingerprints, to minimize and bound the detection latency.

In Figure 3, architectural fingerprints reach complete coverage of architectural state within the comparison interval (assuming no aliasing). By directly comparing architectural state before retirement, architectural fingerprints bound the detection latency to the fingerprint comparison interval, a parameter chosen by the system designer. Bounded detection latency is possible because architectural fingerprints directly observe retiring values. Furthermore, the time required to generate an architectural fingerprint hash is minimal. Chapter 4 shows how efficient hashing circuits can update the signature in one processor clock, even in wide-issue superscalar processors.

Comparison Bandwidth. The comparison bandwidth counts the number of bits per cycle required to compare the execution of two units. This metric is critical to the system's overall feasibility and implementation cost and is governed both by the comparison granularity and by the comparison interval.

Consider systems that directly compare execution state updates. At coarse granularities, such as the chip-external boundary, the required bandwidth is far lower than comparing every latch in the design (hundreds of bits versus millions of bits per clock). However, the cost of comparing at coarser granularities is the increased detection latency. Closer to the processor core, the bandwidth required for direct comparison increases dramatically—matching the retirement bandwidth in the architectural register file and L1 cache write port bandwidth (several hundred bits per cycle). This bandwidth can only be sustained over dedicated on-chip datapaths.

An architectural fingerprint observes as much state as direct comparison at the architectural register file and L1 cache write ports combined, but summarizes the whole sequence instructions into a hash of only a few bytes—which need not be compared every cycle. The fingerprint comparison interval has an inverse relationship with the comparison bandwidth. The Reunion study in Chapter 5 shows that an interval of just sixteen instructions is sufficient to allow comparison fingerprints over the on-chip memory interconnect of a modern chip multiprocessor. A prior study shows that intervals of thousands of instructions are acceptable for error detection across nodes in a distributed shared memory machine [105].

Error Coverage. Error coverage measures the fraction of all errors that can be detected before reaching an unrecoverable state. When coverage is imperfect, the processor can suffer from SDC or DUE because they were not detected in time.

Coverage decreases in several ways. First, the aliasing in the detection mechanism itself may cause erroneous state to appear error-free. Second, important state bits may be omitted or ignored in the detection mechanism (e.g., because the state register is difficult to access). Third, the system can mask errors before they reach the detection mechanism. Finally, in spite of the other factors, if the error is detected—but too late to isolate and correct it—coverage is lost. In actual designs, error coverage does need not be perfect, but it must meet a designated error budget [71].

Architectural fingerprints containing errors can alias with architectural fingerprints from error-free execution, causing a loss of error coverage. The hashes used to generate the fingerprint necessarily lose information as a byproduct of compacting a large number of bits—with some bits in error—to fewer bits. Chapter 4 studies efficient hashes that approach ideal aliasing probabilities over a range of error patterns.

Second, for microarchitectural design or physical layout reasons, not all architectural state can be included in the hash. For example, while general-purpose registers are typically written to a central register file, state such as the program counter, stack pointer, and condition code registers may be written to special, individual registers or at times avoided entirely as an architectural optimization. Fortunately, much of this state indirectly appears as part of other architectural state, and is therefore preserved in the fingerprint.

2.3.1 Discussion

Next, this thesis discusses the interaction of the three metrics through the fingerprint comparison interval and two additional system-level requirements for error detection mechanisms.

Fingerprint comparison interval. The fingerprint comparison interval is the distance, measured in instructions between successive architectural fingerprint comparisons. The fingerprint summarizes all monitored state updates within the interval. Depending on the context, the interval can be a constant or application-dependent. For example, non-idempotent operations such as I/O can force an architectural fingerprint comparison before the end of a periodic interval. The detection latency is bounded by the comparison interval because the fingerprint summarizes all monitored architectural state updates within the interval. To the first order, comparison bandwidth is inversely proportional to the fingerprint comparison interval because only one fingerprint must be compared for each interval. Finally, error coverage depends on timely detection. If results are not held within the core prior to checking, an excessively-long interval can allow an undetected error to propagate outside of the processor and to an unrecoverable location. By contrast, if the results are held, a long comparison interval can incur a performance loss. These tradeoffs are discussed further in the context of the Reunion execution model in Chapter 5.

There are also two important system-level requirements important to error detection in this work, but independent from the metrics listed above.

Determinism. For architectural fingerprints, a processor is considered architecturally deterministic if it always performs the same operations and produces the same architectural outputs for a given sequence of inputs and initial conditions. This is a stronger requirement than functionally correct execution—for a given set of inputs, multiple possible architectural outcomes are correct depending on timing and the outcome of “undefined” outputs [44]. However, architectural determinism does not mean that the microarchitecture must perform precisely the same operations on a cycle-by-cycle basis, as in lockstep. Instead, only the architectural outputs must be deterministic, while the underlying microarchitecture may operate differently, or even be implemented differently, across redundant executions.

Architectural determinism is a concern because a hash-based error detection mechanism cannot discriminate between two architecturally different, yet both correct, outputs and an output containing an error. In both cases, the detection mechanism signals a potential error. In a system with only detection, this behavior decreases the effective reliability of the system by triggering on an error that does not exist. In a system with recovery, this behavior incurs unnecessary recovery operations, which affects performance and potentially leads to problems with forward progress. The latter trade-off is exploited in Chapter 5 with the Reunion execution model.

At-speed operation. Second, the error detection mechanism must work at-speed. Systems with runtime soft error detection cannot run more slowly than systems without detection because they must still meet aggressive performance goals in order to be competitive in the marketplace. Early wearout detection must also work at-speed because errors initially appear from timing faults that are only activated when running at the highest operating frequencies. Therefore, both applications require the error detection mechanism to run at-speed.

2.4 Conclusion

This chapter introduced the fault model and important terminology for this thesis. The concept of the architectural fingerprint was defined and the metrics for evaluating error detection mechanisms were presented. The following two chapters provide concrete discussion on architectural fingerprint implementation, discussing the mechanics of collecting, assembling, and comparing fingerprints in Chapter 3 and the design for a scalable architectural fingerprint hash unit in Chapter 4.

Chapter 3

Architectural Fingerprint Implementation

Architectural fingerprints have a rich interaction with the instruction set architecture and microarchitecture designs. The hardware capturing an architectural fingerprint must match the retirement bandwidth of aggressive modern superscalar, out-of-order processor designs, yet contend with the burdens of complicated instruction sets and highly optimized microarchitectures.

This chapter explores the hardware design and implementation of architectural fingerprints in two commercial microprocessor designs. This chapter assumes an understanding of the architectural fingerprints described in Chapter 2. This chapter is organized as follows. Section 3.1 presents a trace-based proof-of-concept implementation of architectural fingerprints in a superscalar out-of-order processor. Section 3.2 presents an RTL architectural fingerprint implementation and redundant execution in a multicore, multithreaded scalar pipeline microarchitecture. This chapter concludes with brief synthesis results for the architectural fingerprint unit and a statistical error injection study to demonstrate the effectiveness of architectural fingerprints for detecting errors in architectural state.

3.1 Architectural Fingerprints in a Superscalar Out-of-Order Core

This section describes an investigation of architectural fingerprints in a commercial superscalar, speculative, out-of-order processor core design. The investigation includes a proof-of-concept de-

Table 1: State in the basic IA-32 environment that is covered directly or indirectly by an architectural fingerprint.

Class	State	Captured
Integer	8 general purpose registers	Directly covered, except ESP
	EFLAGS	Partly covered, mask undefined fields
	Program counter (EIP)	Indirectly covered
Floating-point	8 general purpose registers	Directly covered
	8 MMX registers	Directly covered
	8 XMM registers	Directly covered
	CR/SR/TR/MXCSR status registers	Directly covered
	x87 opcode, FIP, Data PTR registers	Not covered
Segment	6 segment registers	Directly covered
Memory	Store addresses and value	Directly covered

sign and discusses the architectural and microarchitectural issues that only become apparent when applying the fingerprint concept to an actual microarchitecture (Intel P6).[1]

3.1.1 P6 Overview

This section begins with an overview of the P6 microarchitecture that is relevant for capturing architectural fingerprints. The IA-32 architectural state contained in the P6 which must be captured in an architectural fingerprint is listed in the IA-32 Intel Architecture Software developer's Manual [44] and summarized in Table 1. The table also indicates whether the state can be captured directly or indirectly by an architectural fingerprint. Due to its complexity and decreasing relevance, the x87 floating-point unit is disregarded in this study. However, floating-point values in the modern MMX and XMM architectural registers are covered.

The P6 processor core is a three-wide retirement superscalar, out-of-order IA-32-compatible core, originally shipped as the Pentium Pro [94]. The most recent Core 2 microarchitectures are for the purposes of this study similar in design, except that Core 2 can retire four instructions per cycle. The core speculatively fetches and decodes CISC instructions, in order, into a sequence of RISC-like micro-ops, executes the micro-ops in a superscalar out-of-order core, and retires up to three micro-ops in program order to an architectural register file in each cycle. Micro-ops write

[1]This section describes work done while the author had access to RTL models and internal validation tests at Intel Corporation for the later-released dual-core designs of the mobile Intel P6-based microarchitecture (Yonah) and the Intel Netburst microarchitecture (Cedarmill) designs. The discussion in this section is based on the P6 microarchitecture, as described by Shen and Lipasti [94].

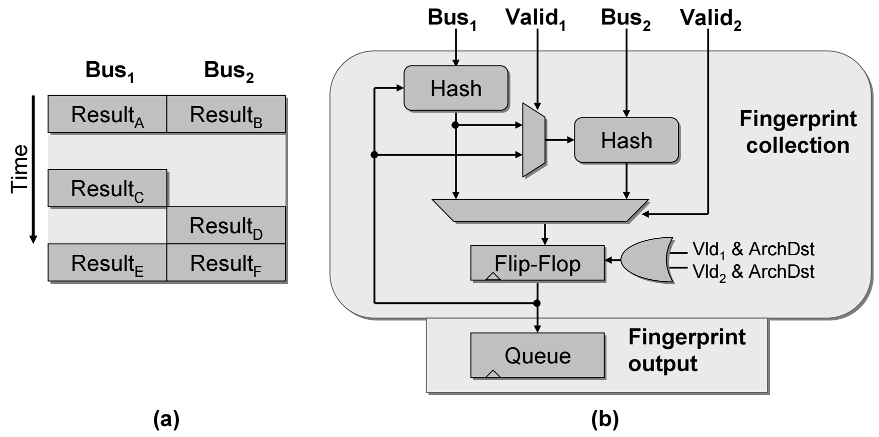


Figure 4: (a) The instruction retirement bus can retire ordered combinations of instructions. (b)The corresponding architectural fingerprint unit.

results to an entire 32-bit register or a portion of the register for legacy instructions. In each cycle, a single store can be written to the store buffer and another store can be committed non-speculatively to the cache. The state in Table 1 is a subset of the total state stored in the architectural register file because there are also temporary registers—registers that are not architecturally-visible—used by micro-ops to execute complex instructions.

Integer, control flow, and load instructions retire values in program order to the integer architectural register over a three-wide retirement bus. Fortuitously, the microarchitecture also guarantees program ordering across the three possible micro-ops retiring in a given cycle because the ROB operates as a FIFO [94]. Alternatively said, any combination of retirement buses can retire architectural results, however the buses are always ordered such that the oldest results always occur on the lowest bus numbers. This greatly simplifies collecting program-order results, as compared to a bus that allows any ordering within the cycle. The retirement combinations over time are illustrated in Figure 4(a), simplified for presentation purposes, to an equivalent two-wide pipeline. On a given cycle, both, one, or none of the buses retire architectural results. New program counter values (EIP in Intel parlance) and condition code values (EFLAGS) are also generated for each retiring micro-ops. This means most values necessary for architectural fingerprints are already available. Valid and destination architectural register number signals exist which identify both when and to which registers a value is being written.

The store buffer has dedicated read and write ports. Values are written back out-of-order, but retired from the store buffer to the L1 data cache in program order. Physical addresses are also written to the store buffer.

The architectural fingerprint for superscalar values is captured using an architectural fingerprint unit depicted in Figure 4. The figure is simplified for presentation purposes to a two-wide retirement. The hash unit consists of combinational logic, described in detail in Chapter 4. The multiplexer logic ensures that values are collected and hashed in program order. The illustrated logic depth is comparable to the retirement stage selection logic for the final program counter and flags registers, which indicates that the fingerprint unit can have cycle time requirements similar to existing logic.

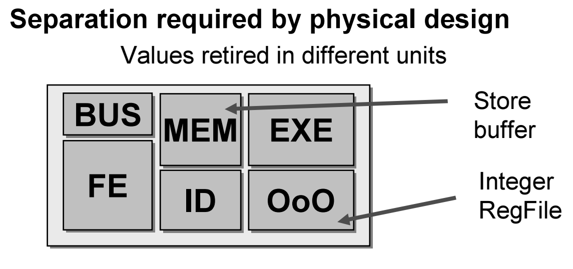
3.1.2 Architectural Fingerprint Constraints

From the above description, implementation of architectural fingerprints appears straightforward. However, the actual implementation runs into several complications—none insurmountable—outlined below. These complications required the implementation of multiple independent fingerprints, based on instruction class, and the addition of simple masking logic.

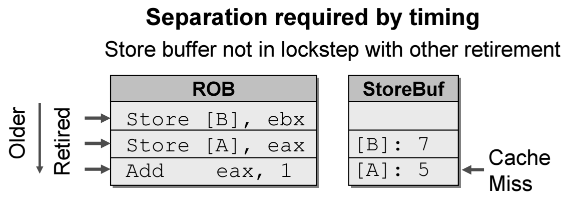
Physical design. The initial concept of an architectural fingerprint called for a single hash of all architectural state. However, the physical design of the core makes this task difficult. The core's floorplan determines how easily various parts of architectural state can be collected and combined in an architectural fingerprint. If all state is nearby, collecting the state together poses few problems. However, processors are physically constrained and retiring data is distributed throughout the processor. For example, in the illustration in Figure 5(a), integer value retirement occurs in the execute unit, while store values are retired in the memory unit. These units can be distant from each other and communication between them requires long, slow global wires.

Instead, independent fingerprints for each output class are much better-suited to addressing the physical design constraints. Related outputs classes are typically stored close together, for example integer values, floating-point values, store values and addresses, and ancillary state, such as IA-32 segment registers are each self-contained.

The additional bandwidth cost of a few (three to four, depending on the architecture) fingerprints is small compared to the bandwidth savings from amortizing comparison across a fingerprint interval. Furthermore, the additional bandwidth may already come for free. For example, if an on-chip memory interconnect is used for transferring fingerprints, the message payload size may already be


Separation required by physical design
Values retired in different units
BUS
MEM
EXE
FE
ID
OoO
Store buffer
Integer RegFile

(a)



(b)

Figure 5: Architectural fingerprints need to be generated by output class because of (a) physical design constraints and (b) ordering and timing constraints.

optimized for larger 64-bit transfers [113], which allows a handful of fingerprints to be included at no additional cost over a single fingerprint value.

Asynchronous and delayed outputs. The retirement stages of a microprocessor need only provide the illusion of program-order retirement. The actual implementation can write values to durable architectural state out of program order. This occurs on both the architecture level and microarchitecture levels. In IA-32 and most modern architectures, an example of the former situation is that the memory consistency model allows implementations to delay stores from committing to the global memory ordering (through the use of a store buffer), even after subsequent instructions have retired from the reorder buffer (ROB) to the architectural register file. Figure 5(b) illustrates the reorder buffer retiring subsequent instructions, while the store buffer delays committing older stores to cache because of a write miss. An example of the latter situation is that, related values, such as store addresses and values may be read in order, but in different pipeline stages, meaning that the values needed for architectural fingerprints are available, but not necessarily at the precise cycle time—or in the order—desired.

This problem is also largely solved by the same solution as for physical design: separate architectural fingerprints for each instruction class. This solution works because values within the same class are still retired in program order with respect to each other (e.g., integer register values and stores both retire in program order with respect to other integer register values and stores, respectively). The delayed store address problem is solved trivially by adding a staging latch to hold the value until the address becomes available.

Undefined outputs. Some architectural outputs are nebulously defined to be “undefined” in the architecture specification [44] (this designation is not unique to IA-32, however). For these outputs, the retiring value cannot be guaranteed to be the same from one execution to another of the same program. This affects architectural fingerprints because an undefined value can cause two fingerprints to mismatch, even when no errors are present. In IA-32, a number of EFLAGS fields are undefined for several integer arithmetic and logical instructions. If a microarchitecture provides a consistent output for these fields then the fingerprint will always match. However, if there are situations where an undefined field depends on internal non-architectural state (i.e., timing-specific microarchitectural state or values on the bus, such as some architecture-specific registers), these values cannot be dependably captured in a fingerprint.

In P6, the values for these “undefined” fields are well defined in the RTL implementation. Therefore, if architectural fingerprints are compared solely between two identical microarchitectures with predictable outputs, the problem is eliminated. However, if the architectural fingerprint is compared against another reference that generates a different value, the undefined value problem must be addressed. This issue is encountered with the architectural co-simulator for P6 (a C program that validates the architectural results of the RTL model). If the entire EFLAGS register is used for an architectural fingerprint, the fingerprints generated between these two models will differ, despite both executions being legal.

A solution to this problem is to mask known undefined fields from the architectural fingerprint. This requires control logic to detect the condition and mask the aberrant bits to produce a predictable value. This solution fails if the undefined value is used in subsequent instructions through program dataflow. However, any program that does this should not be expected to work. [2]

Microarchitectural Optimizations. Optimizations for power and performance can eliminate frequent operations performed in a microarchitecture, which while architecturally defined, do not need to be strictly maintained in the microarchitecture. For example, the IA-32 architecture has a limited number of architectural registers and therefore makes extensive use of the stack pointer (known as ESP) to push and pop local values between the architectural registers and the stack.

[2]The Intel developer’s reference states “Developers must not rely on the absence of characteristics of any features or instructions marked ‘reserved’ or ‘undefined’.” [44].

Table 2: Situations where ESP is consumed or updated and the corresponding detection scenarios.

Original error	Condition	Outcome
ESP as source for load address	Wrong value is loaded from the wrong address	Detected
	Correct value loaded from the wrong address	Undetected
ESP as source for store address	Store address incorrect	Detected
ESP as source for data processing instruction or store value	Propagated to arch. reg or store value	Detected
	Masked in arch. reg or store value	Undetected
Incorrect value written to ESP	Arch. fingerprint created for explicit write	Detected

In recent implementations, the ESP is only updated in limited circumstances—not always when architecturally defined and sometimes only due to microarchitecture-specific timing conditions. Therefore, when blindly observing at retiring register values, updates to the ESP can appear non-deterministic. Recent P6 implementations contain a dedicated stack pointer engine to the pipeline to improve power efficiency and reduce micro-ops per macro-instruction within the pipeline (Gochman describes this mechanism [38]). In the original P6 architecture, every macro-instruction which updates the stack pointer incurs one micro-op to do so, even if the result is never used. This optimization removes most of these micro-ops by calculating the new stack pointer at decode using a small adder in a dedicated stack engine. The architectural register file is only updated with a micro-op only when an instruction architecturally needs to read the ESP, when the register is explicitly changed by the programmer (e.g., by a move to ESP), or when the pipeline needs to restart (e.g., on mis-speculation and traps). These situations can only be fully determined dynamically.

The solution to this microarchitectural optimization is similar to the masking solution for undefined values. The microarchitecture allows inference of whether the ESP is updated by an injected micro-op or by an instruction explicitly specified by the programmer. Therefore, the fingerprint can dependably include the latter case, but ignore the former. This can cause a loss of error coverage in cases enumerated in Table 2. However, the cases where errors in the ESP cannot be detected indirectly are also cases where the ESP value is already masked, and therefore the error can be derated. The last case can be identified with existing microarchitectural signals and safely captured in the fingerprint.

Variable output widths. The values written to architectural registers and memory are sometimes a different size than the processor's native width (for example, 64-bit processors are becoming more prevalent, yet they continue to run programs that operate on 32-bit integers and byte-sized string elements). Most architectures provide an interface for these narrow stores to memory; IA-32 also allows writes of narrower widths to architectural registers.

These varied widths generally do not present a problem with respect to architectural fingerprints. As with undefined outputs, masking logic can be used to zero-extend the missing output bits to match the native machine width.

Micro-op Ordering. Finally, because the P6 implements CISC instructions—which may have multiple outputs—as a sequence of micro-ops, the retirement order of the constituent micro-ops in an instruction matters for architectural fingerprints. If the ordering of micro-ops for the same complex instruction differs from implementation to implementation or even within a single implementation, the architectural fingerprints will mismatch. The microcode is read from ROM tables. Therefore, in general, the microcode outputs values in a predictable order (in spite of the myriad instruction variants and operating modes). This issue proves not to be a problem in practice.

3.1.3 Pentium 4 Architectural Fingerprints

Architectural fingerprints for the rapidly-disappearing Netburst (also known as Pentium 4) microarchitecture were also briefly investigated.

The most significant difference between the P6 and Netburst microarchitectures, with respect to architectural fingerprints, is in the retirement procedure. Instead of writing retired register values to an architectural register file in program order, Netburst speculatively writes architectural register values to a physical register file that is accessed through an architectural to physical register map table [20]. At retirement, only the updated register mapping is written, in program order, to an architectural register map table. However, the values always remain in the physical register file. Furthermore, because the machine speculates aggressively, the values written to the physical register file are frequently re-written during "replays" of a speculative instruction. Furthermore, if the instruction is on the wrong-path, the final register mapping may never be retired.

This speculative writeback means that architectural fingerprints cannot be implemented in the existing Netburst microarchitecture without significant changes. Nowhere in the microarchitecture can architectural values be observed in program order. In principle, architectural fingerprints can be constructed in Netburst and other physical register file-based machines in the same way as for P6; however, the costs are prohibitive. To construct an architectural fingerprint, the values must be read out of the physical register file at retirement. To avoid impacting performance through register file port contention, this solution needs additional register file read ports that match the retirement width of the processor. This is an expensive addition in area, timing, and power to an already highly ported structure [34] and therefore makes implementing architectural fingerprints expensive in Netburst.

3.1.4 Evaluation

This section presents the evaluation of the trace-based proof-of-concept architectural fingerprint implementation on a P6 RTL model.

Methodology

The P6 architectural fingerprint experimental setup consists of a single-core full-chip Yonah RTL model, including L2 cache and external memory, Perl-based trace collection tools, a modified version of the C-based functional x86 architectural co-simulator called `archsim`, and off-line trace analysis tools.

The RTL model loads and executes compiled memory images of assembly-based functional validation test programs, following a brief processor initialization and reset sequence. On every cycle during simulation, the trace collection tool monitors hand-selected internal RTL signals, latches retiring values, and dumps the raw output, cycle-by-cycle, to a trace file. This models the hardware required to capture architectural values from the actual processor. Separately, the modified architectural simulator also produces an architectural state trace for the same program.

The RTL and architectural simulator traces are processed to calculate architectural fingerprints for each architectural result, separated by physical location into different classes for integer values, floating-point values, store values and address, and x86-specific segment registers. The functional simulator has no concept of timing, while the RTL model produces detailed timing information, so the relative order of each fingerprint class differs as discussed earlier. The trace analysis tool compares sequences of each class separately. The analysis tool reports matches over the full execution of the test program and reports mismatches immediately. No errors were injected in this evaluation.

Results

The simulation methodology outlined above was applied to over sixty focused x86 ISA validation programs, twelve cache validation programs, the dhrystone benchmark, a paging test with virtual memory enabled, and a suite of hand-written assembly programs.

In all cases, the architectural fingerprint implementation in both the RTL monitor and archsim matched. This simulation model required numerous revisions, as new instructions and behaviors were encountered. Furthermore, architectural fingerprint also proved to be a highly-sensitive bug detector for its own implementation—whenever a single piece of state was missing or sampled at the wrong time, the architectural fingerprint was clearly different from the architectural simulator and clearly identified a specific dynamic instruction that needed investigation.

This trace-based proof-of-concept demonstrates that the data necessary for assembling architectural fingerprints is available in real superscalar out-of-order microarchitectures. Furthermore, the data can be collected feasibly with modest hardware additions for masking.

3.2 System-level Implementation of Architectural Fingerprints

This section studies the implementation of architectural fingerprints in the multicore, multithreaded, scalar OpenSPARC T1 processor RTL model. This study fulfills several goals. First, this study serves as a substantive demonstration of architectural fingerprints working in a commercial processor design. Second, this study quantifies the coverage of architectural fingerprints for soft errors and demonstrates that they are effective detection mechanisms for silent data corruption. Third, this study demonstrates redundant execution with architectural fingerprint comparison in both multithreaded and multicore designs. Finally, this study explores a system-level implementation where architectural fingerprint values are exposed to higher-level processes.

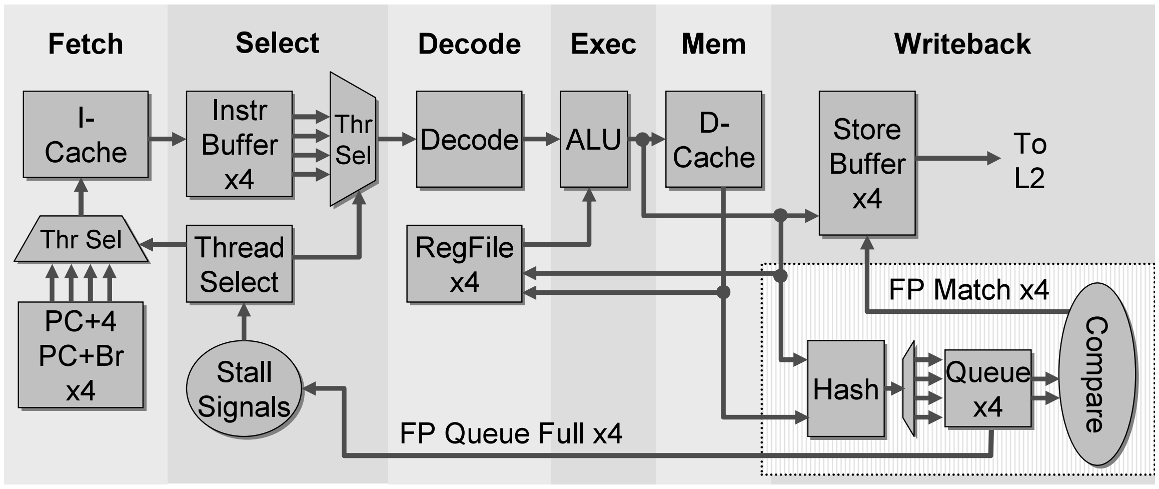


Figure 6: The OpenSPARC T1 6-stage pipeline with architectural fingerprint collection and comparison hardware. Pipeline figure adapted from [113].

3.2.1 OpenSPARC T1 Overview

This section gives an overview of the OpenSPARC T1 microarchitecture, as it relates to fingerprinting. The microarchitecture is simple enough to permit prototyping of architectural fingerprints and redundant execution directly in the RTL model.

The OpenSPARC T1 consists of eight scalar in-order processor cores. Each core selects dynamically from up to four hardware thread contexts on every cycle. The simplified pipeline is illustrated in Figure 6. The portions that relate to architectural fingerprints are now described. In the writeback stage, the pipeline determines if the instruction can retire (or, alternatively, triggered an exception). If the instruction is declared safe, the pipeline writes register values in-order into an architectural file through two write ports. One port is dedicated to values from the ALU and another for so-called "long-latency" operations such as loads and floating-point operations which can take a variable amount of time to complete (the single floating-point unit is shared across all eight cores on the chip). On any cycle, a given thread can have at most one value retiring to the register file, although two different threads can simultaneously write values. Store values and addresses are written into a dedicated eight-entry store buffer for each thread. These values are subsequently written back to the shared L2 cache.

Architectural Fingerprint Prototypes. In this study, three instantiations of architectural fingerprints are prototyped in the OpenSPARC T1 RTL. First, for error injection, an "open-loop" multithreaded implementation of architectural fingerprints is implemented, where an architectural fingerprint is created for each thread, but the fingerprint values are simply logged to a file. As with the Yonah model, architectural fingerprints are accumulated separately for integer, floating-point, and store values. This implementation is useful for error injection studies where the fingerprints can be compared off-line to a golden model (either a error-free OpenSPARC execution or an architectural simulator). The architectural fingerprint RTL prototype is verified against a functional architectural simulator software model for more than 900 single core OpenSPARC verification programs.

The second model is a "closed-loop" model where architectural fingerprints are compared between user-level redundant threads within the same core. This model targets protecting user-level execution from silent data corruption. Architectural fingerprints are queued for comparison, incurring stalls if one thread proceeds too far ahead of its redundant partner thread. Furthermore, the store buffer is modified to gate unchecked stores and prevent them from entering the memory system. User-level stores are only released after a successful comparison with the partner threads' stores. This design provides "fail-stop" protection for detected architectural errors. The architectural fingerprint register state can also be accessed and controlled through privileged code running on the processor. This RTL model demonstrates system-level aspects of architectural fingerprints, as well as demonstrating their use to compare redundant multithreaded execution within a single core, as is employed in many recent microarchitecture proposals [36, 40, 57, 67, 72, 81, 85, 88, 90, 106, 115]. Because fingerprinting is already demonstrated using the open-loop model, further error injection results are not presented for this model.

The third RTL model is a straightforward extension of redundant execution model to execution across two processor cores. Here, fingerprints are transferred across a dedicated cross-core channel to compare the two executions (alternatives to the dedicated channel are discussed in Chapter 5). This RTL design demonstrates architectural fingerprints for comparing redundant execution in a multicore context, as is necessary in recent microarchitecture proposals for redundant execution in CMPs [40, 56, 72, 104, 114] and the Reunion execution model presented in Chapter 5. Because fingerprinting is already demonstrated using open-loop model, further error injection results are not presented for this model.

Table 3. The software interface to internal architectural fingerprint registers.

Instruction	Description
`stxa %l0, [%g1] ASI_ARCHFP`	Store an initial architectural fingerprint value from `%l0` and enable fingerprinting for the thread specified in `%g1`
`ldxa [%g1] ASI_ARCHFP, %l0`	Read the architectural fingerprint value for the thread specified in `%g1` into `%l0` and halt architectural fingerprinting

3.2.2 System-level Design

This section explores several system-level design issues for architectural fingerprints. First, a software-visible interface to architectural fingerprint allows operating systems to enable, disable, and manage the fingerprints and redundant execution. Second, this section discusses the tradeoffs in using virtual versus physical addresses in an architectural fingerprint. Finally, this section introduces tradeoffs and potential applications of fingerprinting all execution state or just user-level state.

Software-visible Interface. A software-visible interface to architectural fingerprint control and data registers allows privileged software to initialize, reset, enable or disable fingerprints. This functionality is necessary for operating system-level error diagnosis, recovery, and reporting. For example, after a soft error affecting redundant execution, the redundant contexts will diverge and their fingerprints will mismatch. Error handling firmware or software (e.g., machine check code or operating system routines) needs to first record and report the error event, then reset the architectural fingerprints to a consistent value in both contexts before recovering and continuing redundant execution.

Additions to the OpenSPARC T1 provide the hooks necessary to stop, analyze, and restart architectural fingerprints within privileged code. This interface is implemented in the RTL model by defining a new address space identifier (ASI) for architectural fingerprint control.[3] The ASI accesses are summarized as SPARC assembly instructions in Table 3. The instructions can address one of the four hardware contexts on the same core by specifying the ASI's virtual address, based at zero and with an offset of eight bytes.

[3]ASIs are a common way to provide memory-mapped access to internal register state in SPARC architectures. The interface is through load and store instructions to an alternate address space.

The store instruction clears any architectural fingerprints queued for the thread and initializes the fingerprint registers to a desired value (16-bit integer, floating-point, and memory fingerprints are simultaneously specified in the store value operand), and enables architectural fingerprinting for the next user instruction retired by that thread. The store also clears remaining, unchecked fingerprints for the thread because one redundant thread can run ahead of the other and therefore queue multiple erroneous unchecked fingerprints before a mismatch is detected. The load instruction halts architectural fingerprint hashing for the specified thread and returns the most recent fingerprint values to the destination register.

This interface has been used in the multithreaded proof-of-concept redundant execution RTL model to compare the executions and identify mismatches in test programs containing two user-level redundant threads.

Virtual and Physical Addresses Store addresses are captured by architectural fingerprints to check for stores being written to incorrect locations. The architect can choose between capturing virtual or physical addresses. Both addresses are readily available in modern processors, so the design choice reduces to tradeoffs in error coverage and application flexibility.

Physical addresses provide higher coverage of the microarchitecture by checking values along the TLB mapping and physical address datapaths ignored by checking virtual addresses only. Without checking the physical addresses, store values can be written to cache silently at the wrong location and latent errors can remain undetected (checking physical addresses is not a complete end-to-end solution for detecting all errors in stores, however much of the datapath beyond translation is already parity or ECC-protected). Furthermore, the operating system frequently uses physical addressing for functions such as page table management and I/O operations. Checking only virtual addresses leaves these critical operations vulnerable.

On the other hand, only checking virtual addresses enables additional applications for architectural fingerprints. In user-level redundant threading [99], two identical processes are started by the operating system with different physical address spaces, but identical virtual address spaces. This simple mechanism permits the two processes to independently store to memory without interfering with each other. By fingerprinting virtual addresses on user code, these redundant processes can be compared using architectural fingerprints: the physical addresses may differ, but the fingerprints still match.

Another application is to check critical sections of user execution, either within a single machine or across machines. Architectural fingerprints can be enabled for a small portion of code and executed twice (sequentially, or in parallel). The granularity can be on the order of a database transaction or other well-defined high-level software operations.

Variants of architectural fingerprints containing both virtual and physical addresses are implemented in the OpenSPARC T1 RTL model. For error injection studies, this work focuses on a fingerprint that covers physical addresses because comparison with a golden model can be done offline (and therefore, no work is needed to make physical addresses match across execution). However, due to the time and effort required to implement a complete system with redundant execution, this thesis demonstrates the redundant threaded execution prototypes using architectural fingerprints only on virtual addresses and user-level state.

User-level Fingerprints. For the redundant execution proof-of-concept, redundant execution and comparison with architectural fingerprints is implemented in the OpenSPARC T1 RTL model for user-level code only. This means each instruction result retired in user-mode is accumulated in an architectural fingerprint, while privileged-mode code is neither redundant nor fingerprinted. For traps, such as TLB misses, the faulting user instruction neither is fingerprinted during its first execution, nor is the trap handling code (which is privileged), however on the user instruction's re-execution the result is accumulated into the fingerprint. Furthermore, the store buffer is modified to gate pending user stores until each has been checked with the redundant thread. Privileged-mode stores, which are not executed in a redundant mode, remain ungated.

The user-level fingerprints give an assurance that the user-level instructions that have been executed are correct. However, this method has the drawback that latent errors in units that are not heavily exercised during user-level execution—such as the trap logic unit—are not captured in the fingerprint. Therefore, while the user-level program executes correctly, errors can accumulate in other regions of the core and will be encountered only when executing privileged code, which is not protected by the fingerprint. In these situations, the operating system can encounter both detected, uncorrectable errors and silent data corruption.

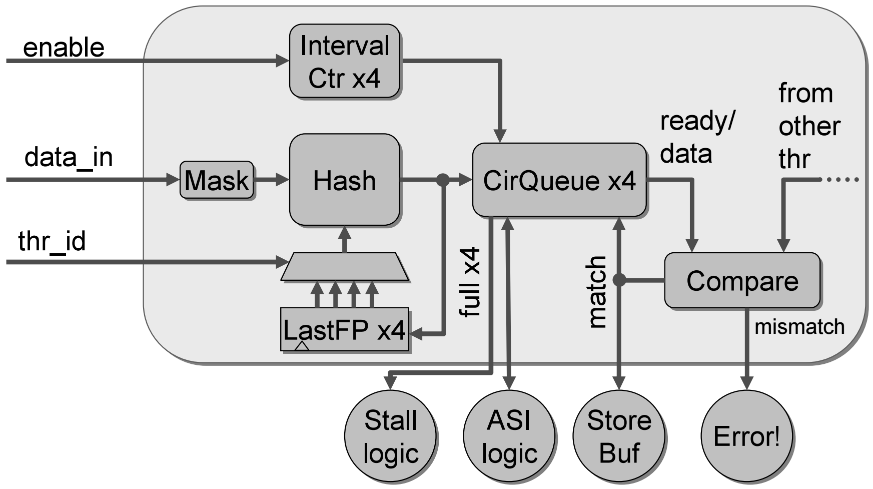


Figure 7. The architectural fingerprint unit in OpenSPARC T1.

While a complete solution for redundant execution in multithreaded user and privileged code is presented as Reunion in Chapter 5, this solution requires checkpoints or instruction rollback. Neither feature is presently available in the OpenSPARC T1 processor. Therefore, for the RTL proof-of-concept, user-level redundant threads are implemented by spawning two identical threads in kernel boot-up code, using identical virtual, but separate physical, address spaces. This, in combination with user-level fingerprints provides a working demonstration of architectural fingerprints in a real processor context.

3.2.3 Hardware Design

This section describes the multi-threaded architectural fingerprint unit that has been implemented in the OpenSPARC T1 Verilog model.

The architectural fingerprint unit consists of three independent collection and hash units, one each for integer, floating-point, and store values and addresses. A simplified architectural fingerprint unit is shown in Figure 7. The inputs are the retiring datapath value, an enable signal and thread ID for the value. In the actual implementation, the enable signal consists of a multiple pipeline signals that collectively indicate when the datapath value is being written (e.g., valid values can still be driven on the bus during failed speculation or during a trapping instruction). Stores values are also masked to the appropriate width.

On each cycle, the hash unit computes a new fingerprint hash value based on the last hash value for the retiring thread (stored in a register) and the retiring value. An incrementer counts the fingerprint interval and inserts the fingerprint in a queue (implemented as a circular buffer) once the fingerprint interval has been reached. The queue contains sixteen 16-bit entries in this implementation, which allows enough buffering for a fingerprint interval of one instruction, while allowing a single thread to fill all pipeline stages, and enough lookahead to stall the thread in fetch before the buffer fills without discarding useful instructions. If the queue fills beyond a high watermark, the queue asserts a stall signal in the thread switch logic to prevent the thread from overflowing the fingerprint queue. In this design, architectural fingerprints only incur a performance impact when the relative progress of two redundant threads is more than ten fingerprint intervals apart.

The architectural fingerprints are compared when two paired threads (statically determined in this proof-of-concept) have valid fingerprint value at the heads of their respective queues. The fingerprints are transferred to a comparator and if they match, the queues free the fingerprint entry. Furthermore, for store fingerprints, the store buffer is notified that it may release the stores from the fingerprint. To support this operation, the store buffer required twelve additional bits of state to track gating and the user/privilege level of each store value. Upon a mismatch, a global signal triggers the threads to stall and the simulation halts (a real processor can instead initiate a trap handling routine).

The same architectural fingerprint unit is used for both redundant multithreading within a single processor and for checking redundant threads across processor cores. The proof-of-concept with redundant threads has been validated with soft error injection experiments. Depending on the error, the processor either is forced into a fail-stop state or has already deadlocked. In the former case, the fail-stop mechanism both prevents unchecked user stores from entering the memory system and stops further execution. The latter requires a reset sequence for recovery.

3.3 Architectural Fingerprint Synthesis

This section briefly evaluates the synthesized area utilization of the complete user-level architectural fingerprint unit in relation to the remainder of the OpenSPARC T1 core. The processor core and architectural fingerprint units are synthesized using Synopsys Design Compiler 2005.09 mapping to the Artisan/TSMC 0.18um low-power standard cell library [9]. Due to a lack of a memory compiler, only combinational and sequential logic, but not memory arrays, are evaluated. The architectural fingerprint unit is multithreaded and contains three independent hash circuits for integer, floating-point, and memory fingerprints, respectively. The hash circuits are based on the X-compact [69] spatial and MISR temporal compaction design described in Chapter 4.

The baseline OpenSPARC T1 processor core occupies a total of 3,209,187μ^2 (excluding memory arrays). The additional overhead of the architectural fingerprint unit is 118,493μ^2, or an additional 3.7% area overhead. This area overhead is well below the 10% rule-of-thumb area overhead for architectural reliability mechanisms in commodity processors [29]. Furthermore, the OpenSPARC T1 has an extremely area-efficient "lean" scalar pipeline design, which magnifies the cost of the architectural fingerprint unit. In aggressive superscalar out-of-order microarchitectures, such as the Intel Core 2, the area overhead is commensurately smaller. In exchange for this overhead, later chapters of this thesis show that architectural fingerprints provide both soft error and early wearout detection.

3.4 Soft Error Injection Evaluation

This section evaluates the coverage of radiation-induced soft errors for architectural fingerprints proof-of-concept in the OpenSPARC T1 RTL model.

3.4.1 Methodology

Architectural fingerprints are evaluated using statistical soft error injection in the single-core OpenSPARC T1 RTL model with the open-loop architectural fingerprints described earlier in this chapter. Synopsys VCS version Y-2006.06 simulates the Verilog model with custom Verilog PLI modules added for soft error injection. This experiment examines the detection capabilities of architectural fingerprints in isolation, without regard to recovery. The error detection in this experiment is sufficient for fail-stop systems, while issues related to recovery are explored further in Reunion.

For each workload, the RTL model is first executed in a error-free environment to establish the error-free fingerprint values. The model is then run repeatedly with statistical soft error injection in pipeline latches. Latches are specifically targeted because they are numerous enough to affect reliability and, unlike SRAM, cannot easily be protected by ECC or parity.

In each run, the processor boots and starts the test program. Once the program starts, the error injector injects a single bit flip into a latch in the selected unit at the specified cycle. The upset bit is selected uniformly across all latch bits in the unit. The injection cycle time is varied uniformly across twenty possible points in execution. The model continues to run until completion of the program (detected by reaching a specified completion program counter on all threads) or a timeout detected by 10,000 cycles of inactivity on any thread. The latter situation indicates that the error caused the core to deadlock.

This experiment models the expected impact of radiation-induced upsets on latches (because of the relatively large area of a latch, compared to an SRAM cell, multi-bit upsets are currently considered unlikely). A single upset does not imply, however, that only one bit will be affected in architectural state: the upset can either be masked or propagate to one or more latches. This experimental methodology does not detect latent errors—errors that are neither masked nor propagated to architectural state—in the processor core at the end of program execution. With respect to execution of the test program, latent errors are considered masked, however other test programs or further execution may eventually expose them.

The module injects errors into several top-level and representative units in the OpenSPARC T1 design. The units are briefly described below:

- `byp` is the operand bypass network and part of the execution unit, which includes ECC generation for architectural register file writes.
- `exu` is the execution unit, including control paths and ALU datapaths.
- `fcl` is the instruction fetch control logic, which handles instruction caches misses, PC and branch computation.
- `fdp` is the fetch datapath controlled by the `fcl`; it is responsible for holding all PC and computing next PC values and branch outcomes for all pipeline stages.
- `lsu` is the load-store unit, which comprises datapaths and control logic for load operations and the store buffer. `swl` is a representative set of finite state machines which control thread selection.

Table 4. The test programs used in soft error injection.

Name	Dynamic instructions	Test Description
`dram_mt_4th_loads _attrib_many`	3,610	DRAM load/store misses
`exu_irf_local`	39,538	Local windowed registers and bypass network
`mt_alu_ldx`	1,264	Combination of ALU, load, and endian programs
`mtblkldst_loop`	2,564	Back-to-back block loads/stores
`mt_Ifill_L2`	1,582	I-cache fills/misses
`mt_raw`	2,018	Combination of read-after-write programs
`tr_tixcc0`	4,232	Integer condition code traps

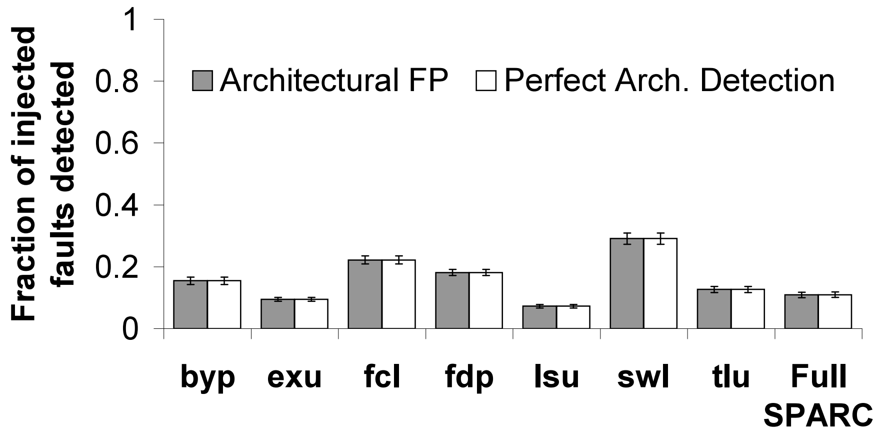


Figure 8. Soft error injection detection results.

- `tlu` is the trap handling unit and is critical for handling translation lookaside buffer (TLB) misses and other common traps.
- The entire SPARC processor core includes the above units plus a stream processing unit, built-in self test (BIST), crossbar staging latches, and the architectural fingerprint unit.

Each unit is exercised with seven multithreaded validation test programs from the OpenSPARC T1 package. These programs, summarized in Table 4, are selected to exercise a wide range of units and processor behaviors. Each unit has at least 1,100 individual soft errors injected.

3.4.2 Results

Figure 8 shows the baseline coverage of architectural fingerprints using the methodology outlined above. Each bar indicates the fraction of soft errors injected that are eventually detected as architectural errors. Architectural fingerprints reveal a high degree of masking. In the full core error injection simulations, 90% of injected bit flips are masked architecturally. This is in rough agreement with prior studies that report high degrees of architecture-level and program-level masking [32, 49, 117], ranging from 60% to 90% in unprotected processors and 99.97% of the time in the heavily protected POWER6.

The figure also shows a perfect architectural error detection mechanism, performed by directly comparing the error injected architectural state outputs with a error-free execution. The difference between the perfect detection mechanism and architectural fingerprints establishes the degree of aliasing in the architectural fingerprint. In the presented experiment, 10,629 errors were injected and a single instance of aliasing is observed. With the 16-bit fingerprint in this system, which aliases with a probability of $2^{-(p-1)}$ in the presence of an architectural error, this level of aliasing is expected. Because the architectural fingerprints and perfect architectural detection differ within the bounds predicted by aliasing, this result shows that architectural fingerprints provide an effective summary of the full architectural state. By contrast, a single instance of aliasing cannot establish rigorous statistical evidence of the overall aliasing probability. The aliasing probabilities are instead explored in Chapter 4, where the retirement stages are modeled with a range of architectural error patterns over millions of experiments.

Figure 9 explores, in more detail, the instances where errors are detected by architectural fingerprints. This result disambiguates which failure modes the architectural fingerprints detect. There are three possible outcomes. First, a fingerprint *mismatch* indicates an architectural error—in an otherwise unprotected pipeline, this results in SDC. Second, an *underrun* indicates that the soft error caused the processor to execute fewer instructions than expected without first causing a mismatch, ending in a deadlock. This situation is trivially detected with a timeout mechanism, and therefore, does not produce SDC. The architectural fingerprint unit can detect this situation by observing an architectural fingerprint generated by one execution, but missing from a redundant execution. Finally, an *overrun* indicates that the processor executes more instructions than expected, generally because the processor entered an unexpected loop. Architectural fingerprints detect this situation as SDC at the end of a fingerprint comparison interval. No architectural errors in Figure 9 remain undetected.

For all units except the thread select logic (`swl`), fingerprint mismatches dominate the errors

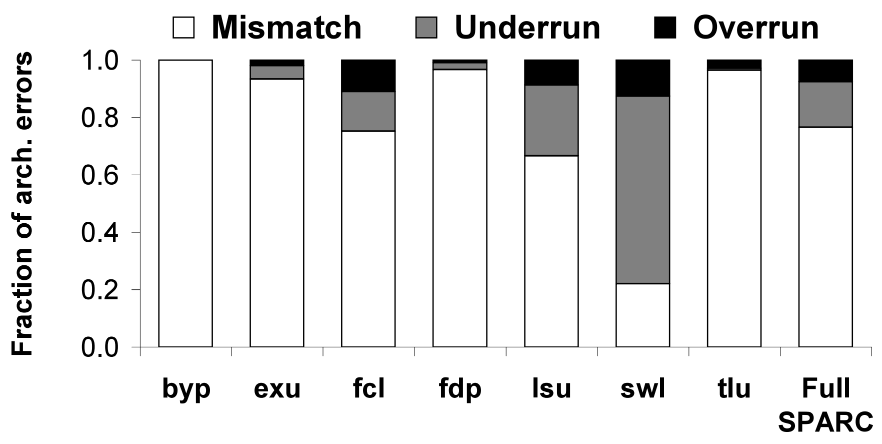


Figure 9: Outcomes in instances where architectural fingerprints detected an architectural error.

in architectural state. In the extreme case of the bypass unit (`byp`), all errors first caused silent data corruption. The bypass unit contains pure operand datapaths; therefore silent data corruption is expected. By contrast, the thread select logic is instead dominated by underruns. Underruns happen here because an error in the thread switch logic typically causes the sparsely encoded finite state machines to enter invalid states. In an invalid state, the thread will never be considered "ready" and therefore the thread stops executing entirely.

Overall, this result shows that the majority of architectural soft errors manifest first as SDC. Architectural fingerprints can detect all three failure modes; however, unlike simple detection mechanisms such as timeouts, architectural fingerprints provide timely detection of SDC.

3.5 Conclusion

This chapter explored the hardware design and implementation of architectural fingerprints in Intel Yonah and Sun OpenSPARC T1. Both designs demonstrate that dependency hurdles lurk in the instruction set and microarchitecture designs; however the proof-of-concept designs show that none of these hurdles were insurmountable. Finally, the chapter demonstrates, through RTL soft error injection, that architectural fingerprints effectively detect architectural errors.

Chapter 4

Hash Design

4.1 Introduction

This chapter explores the design tradeoffs of different architectural fingerprint hash architecture. The prior chapter assumes an X-Compact [69] tree and multiple input shift register (MISR) for the architectural fingerprint hash design. This chapter provides the analysis to support that design choice.

The problem addressed in this chapter is how to compact efficiently architectural state outputs into a hash. Both architectural fingerprints and traditional manufacturing test techniques, such as scan chains, require high degrees compaction to compare efficiently a complex design unit with a reference. Because traditional scan chain compaction has been well-studied in the context of offline manufacturing test, this thesis takes inspiration from manufacturing test techniques for architectural state compaction.

However, the architectural state compaction problem differs from manufacturing test in two important ways. First, the compaction must be applied to a discrete instruction results (e.g., an ordered sequence of 64-bit values), instead of a continuous stream of values from an array of scan chains. In scan chains, when latches in a design change, the number and order of scan latches may also change. By contrast, architectural fingerprints must be able to construct the same hash value for a program's execution, regardless of the underlying microarchitecture and implementation. Therefore, the hash architecture must be specifically designed to preserve this organization.

Second, the compaction latency is critical for architectural fingerprints. In scan chains, tester time should be minimized to reduce costs, but scan chains frequently have latencies of thousands of cycles; however, because comparison is performed offline, there is little hardware or performance cost to increasing this latency. By contrast, architectural fingerprints are used in an online error detection context where the detection latency is tied to critical architectural features such as fail-stop operation or checkpoint-based rollback recovery. These features demand latencies of only a few cycles for timely error detection. Therefore, the online nature requires a hash design that meets this latency.

The hash architecture requires careful design to ensure that (1) the aliasing rate—which directly affects error coverage—is acceptable for the expected error classes, (2) the area and latency overheads are reasonable, and (3) the architecture can scale to support the retirement bandwidth from narrow single-issue to wide superscalar pipelines.

An effective architectural fingerprint hash preserves evidence of errors in retiring architectural state. Because a single bit upset can cause one or more bit errors in the final retiring state [54, 91] and errors can propagate by program dataflow to other instructions before detection [105], the hash must be effective for a range of error patterns over both space and time.

However, the design must fit with reasonable area bounds with respect to the processor core it monitors. Ten percent is rough yardstick for the acceptable total area overhead dedicated to reliability mechanisms in commodity microprocessor designs [29]. The architectural fingerprint hardware is only one component of the possible reliability mechanisms (e.g., other mechanisms include parity and ECC on caches, control for redundant execution, etc.) and therefore can only consume a fraction of the reliability budget. Furthermore, logic for reliability must perform within the clock frequency goals specified by the design. Therefore, a fingerprint architecture must also have a latency less than or equal to existing pipeline logic.

Finally, to avoid becoming a performance bottleneck, the architecture must match the sustained retirement bandwidth of modern pipelines. The hash is accumulated across instruction results, but the operation is not commutative (to detect reordering of results). A hash architecture must be flexible enough to work with a range of microarchitectures that are used in the critical enterprise server space “fat” (e.g., the four-wide superscalar Power5 and Intel Core) and “lean” (e.g., the scalar Sun OpenSPARC T1) .

The ideal hash design from an aliasing perspective is the cyclic redundancy check (CRC). While parallel-input CRC implementations are known, they either require a deep pipeline to meet the cycle time or grow considerably larger and slower as the input width increases [116]. This chapter shows that simpler hash mechanisms can provide equivalent aliasing protection to a CRC, without incurring the area, latency, and complexity costs of a full CRC implementation.

This thesis proposes an architecture that satisfies these goals and explores several options for the hash hardware design. The following contributions are made:

- A scalable architectural fingerprint hash architecture.
- Empirical analysis of the aliasing properties of candidate compactor designs using error injection and circuit overhead analysis using synthesis to an ASIC standard cell library.
- The observation that while MISRs have poor aliasing properties for few-bit errors, well-chosen spatial compaction trees can effectively amplify few-bit errors into many-bit errors. Thus, the MISR's good many-bit error performance is maintained across different error patterns.
- A practical instantiation of the hash architecture utilizing an X-Compact-like compaction tree and combinational MISR circuit with superior qualities for aliasing, area, and latency compared to the other designs considered.

4.2 Background

The input to an architectural fingerprint hash circuit is an ordered sequence of retiring instruction results. The instruction results are assumed to consist of 64-bit words, as is typical of high-performance microprocessors today. The sequence has a finite length, which corresponds to the fingerprint comparison interval and words within the sequence can contain errors in the form of bit flips. This sequence is hashed into a fingerprint value (assumed to be sixteen bits in this study) which non-uniquely reflects the contents of the original sequence. The goal is for the hash output of a sequence of instructions with errors to differ from the hash of the same sequence of instruction results without errors.

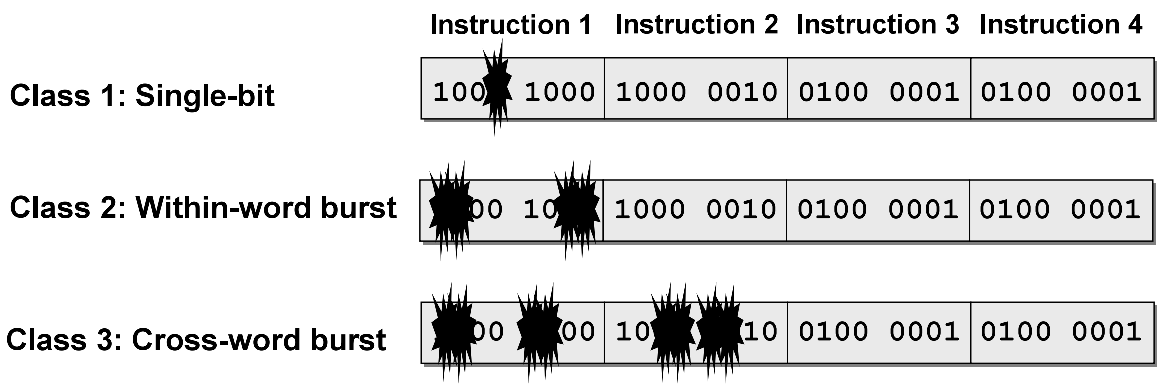


Figure 10: The error classes studied in this thesis, illustrated here with fingerprint intervals of four eight-bit instruction word outputs.

This study first defines an error model to aid in understanding the strengths and weaknesses of the hash circuits. The model is summarized in Figure 10. This error model is approached systematically by separately considering three classes of errors, while avoiding the details of the underlying faults and microarchitecture.

Class 1 errors are single-bit flips that occur within a single instruction word in the sequence. In a processor, this class corresponds to the situation where a single logic bit is flipped in the pipeline, which remains unmasked, and the resulting single-bit difference retires to architectural state without propagating to other instructions. This error class is useful for modeling upsets in pipeline latches, which occupy substantial area, and therefore are likely to experience only solitary bit flips [49].

Class 2 errors are multiple bit flips within a single instruction word in the sequence. This class corresponds to corruption of a single instruction's result from a multi-bit upset event, the corruption of control path logic, or corruption of input operands that remain unmasked in the result. In this class, errors in the output of one instruction do not propagate architecturally to subsequent instruction results. This chapter also distinguishes between *few-bit* errors (2-3 bits in error) and *many-bit* errors (>3 bits in error). Industry expects few-bit errors to become more common with direct datapath corruption from a single-event upsets due to continued scaling, while many-bit errors are more commonly caused by errors in the control path (e.g., redirecting the wrong register value to retirement).

Finally, class 3 errors are bit flips across multiple instruction words. This class corresponds to class 1 or 2 errors that have propagated through dataflow to subsequent instructions in the fingerprint comparison interval. No hashes discussed in this chapter can guarantee detection of all class 3 errors. Nevertheless, some hashes can still provide good probabilistic detection properties over the spectrum of class 3 errors.

4.3 Hash Architecture

This section discusses the design requirements and architecture of the fingerprint hash unit, beginning by examining the properties required and explaining why a simple CRC circuit cannot satisfy all of the design requirements. The section then presents a scalable hash architecture.

4.3.1 Design requirements.

As discussed in Section 4.2, the hash unit must compact a sequence of instruction outputs into a single, small hash value. An ideal compactor has properties similar to a p-bit cyclic redundancy check (CRC) [84], which can detect:

- Any single-bit error
- Any single, double, triple, or odd-bit error in a range up to 2^{p-1} bits
- Any single burst error up to p bits
- Larger-scale errors, with probability of $1 - 2^{-p}$

A sixteen-bit hash output is chosen as the basis for comparison in the remainder of this thesis, based on the results in [105] for the aliasing probability, however the analysis and results here can be generalized to hashes of other sizes.

4.3.2 Parallel Input CRC units.

Unfortunately, building larger CRC units is not without significant costs. With wider pipelines (e.g., the 4-way retirement in recent microarchitectures [7, 8, 46]) directly building a large parallel CRC circuit [5] capable of compressing at least one 64-bit word in one cycle is prohibitive in both area and cycle time. A 64-bit input CRC-16 unit costs more in area than "matching" pipeline components such as an optimized 64-bit adder and exhibits significantly longer latency characteristics. Furthermore, while CRC circuits can be pipelined to meet cycle time requirements, the pipelined

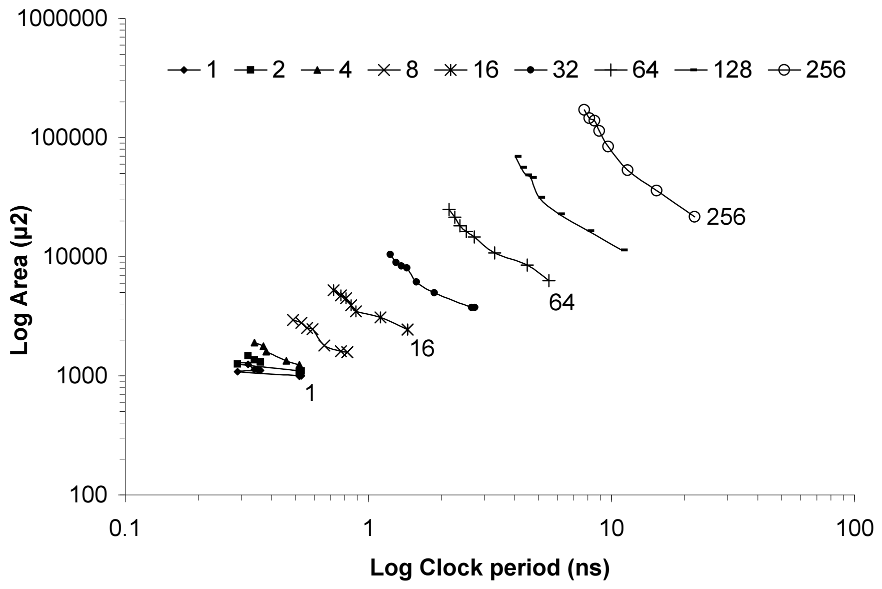


Figure 11: The area-delay curves for parallel N-bit input parallel CRC-16 units. Logarithmic scales are used on both axes to capture the full range.

implementation incurs multiple stages of XOR trees that scale proportionally with the maximum fingerprint interval (in communications terms, this corresponds to the message size) and the area overhead remains. Pipelined implementations are not studied specifically in this work; however, Walma [116] compares the area and latency of pipelined and non-pipelined parallel CRC designs.

The units are synthesized using Synopsys Design Compiler release 2005.09, mapping to an Artisan/TSMC 0.18um low-power standard cell library [9]. The Synopsys DesignWorks library is configured to choose the best adder implementation for the checksum units (therefore, internally the selected adder implementation can change). Mapping efforts for both area and timing were set to "high" and the input clock rate was varied from an empirically determined F_{max} to five times slower, to produce the area-latency Pareto style curves.

Figure 11 shows the cost, over a range of different parallel input sizes for a sixteen-bit CRC computation using the CCITT-16 polynomial: $x^{16} + x^{12} + x^5 + 1$ [47].[1] The single-bit input corresponds to a simple linear-feedback shift register (LFSR), where the cost is dominated by the sixteen latches and the combinational logic comprises only three XOR gates. As the input size reaches sixteen parallel inputs, the XOR network begins to dominate the area and latency costs. Area and cycle time continue to grow linearly with larger input sizes. At 64-bit parallel inputs, the

[1]Other sixteen-bit primitive polynomials that exhibit similar aliasing, area, and latency properties exist. The same general properties hold for other primitive polynomials.

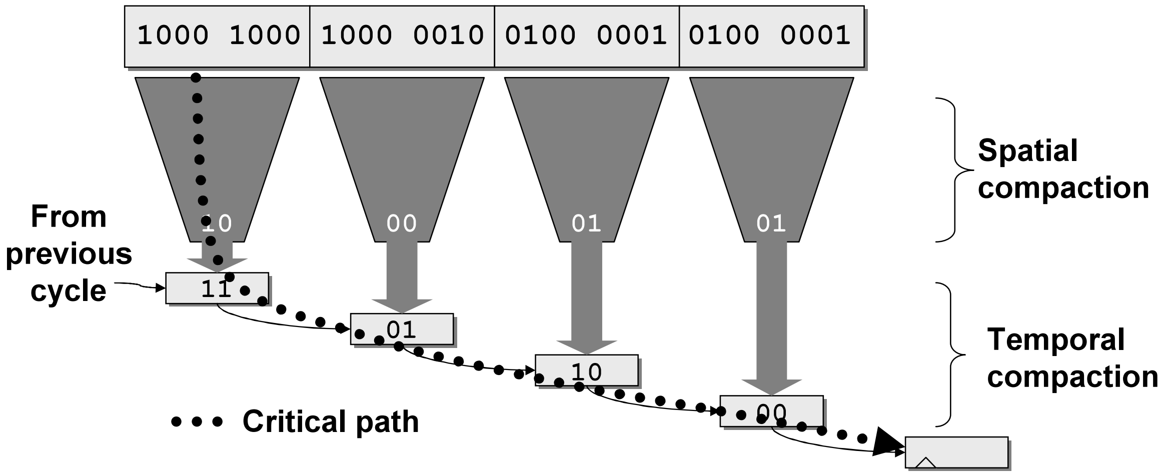


Figure 12: The scalable hash architecture consists of two stages: space compaction and time compaction. Space compaction independently shrinks instruction results, while time compaction shrinks across instructions. The above architecture is shown for a four-wide superscalar pipeline.

cost rivals the best synthesized 64-bit adders, but is several times slower (the reference adders are shown in a later section). At 256-bit parallel inputs, the area and latency is similar to a 64-bit output multiplier implemented in the same process. In general, minimum latency and area grow linearly, but optimized implementations show significantly higher marginal area costs for each additional bit.

The take-away message of Figure 11 is that a direct CRC implementation is expensive in terms of area and is also infeasible because of latency requirements. For scalar pipelines, such as the Sun Niagara T1, a 64-bit input parallel CRC-16 rivals the area of its 64-bit carry-lookahead adder. Therefore, a 64-bit parallel CRC-16 unit has a high cost for scalar cores. For a four-wide superscalar pipeline, the 256-bit input parallel CRC-16 unit must be clocked several times slower than the rest of the core. Therefore, latency requirements mean that a parallel-input CRC is impractical for wide-issue pipelines.

4.3.3 A Scalable Hash Architecture.

This section proposes a hash architecture that can simultaneously meet aliasing, area, and latency requirements. The architecture consists of two stages illustrated in Figure 12: space compaction and time compaction. The space compaction stage is a parity tree-like combinational path that takes results from individual instructions and compacts them to a narrower value. The time compaction stage is a combinational path that first compacts the space-compacted results of each instruction, in sequence, and then stores the result in a latch. A multiplexor (not shown) selects between outputs if only a subset of the retirement buses in a superscalar processor are used in a given cycle.

The basic concept of combined space and time compaction is well known in the field of manufacturing test to reduce comparison bandwidth and response storage requirements [93]. However, these designs are typically applied to compact unstructured arrays of scan chains into a hash. This thesis applies space-time compactor concept to selected architectural state. Unlike manufacturing test, where the meaning of the input state is largely irrelevant to the compactor, this compactor architecture produces a hash of architectural state updates that is carefully constructed to be independent of the underlying microarchitecture.

This section now briefly discusses the aliasing, area, and latency properties of the compactor architecture (aliasing properties can only be truly discussed after introducing properties of the compactors in Section 4.4).

Aliasing. The aliasing properties in this architecture are a function of constituent space and time compaction components. In particular, the overall compactor aliases if there is aliasing in either the space or the time compactors. In general, the overall aliasing probability is not as simple as the calculating the joint probability of the constituent compactors in isolation. The simple joint probability does not hold because the distribution of bit errors changes from the input of the space compactor to the input of the time compactor. Therefore, the aliasing probabilities between the space and time compactors are dependent. This dependence leads to interesting (and convenient) properties that are not be available if independence were preserved. This result is discussed quantitatively in Section 4.4.

Area. Compared to a single, wide parallel input CRC unit the time compaction has significant area savings, because it must only compact one fourth as many bits. In general, the space compactors are also smaller than the portion of the parallel input CRC unit that they replace. The costs are quantified for specific instances in Section 4.5. The area scales with the following equation, where $PipelineWidth$ represents the width of the superscalar pipeline:

$$Area = PipelineWidth \cdot (Area_{space\_compactor} + Area_{time\_compactor}) + Area_{latches}$$

Latency. The latency is improved over a single, wide parallel input CRC unit because the design is amenable to trivial pipelining. The space compactors can be decoupled from the time compaction, and because they occur independently, can be internally pipelined. Parity trees generally are not so deep as to require this and the pipeline latch overhead can eliminate any potential savings. Furthermore, in a superscalar compactor, space compaction occurs in parallel for each instruction. By contrast, the time compaction depends upon the previous cycle's values to compute the current cycle's values. Therefore, the time compaction component's latency increases with the superscalar pipeline width. The latency of an unpipelined compactor architecture, illustrated by the dotted critical path in Figure 12 follows this equation:

$$Latency = Latency_{space\_compactor} + PipelineWidth \cdot Latency_{time\_compactor} + Latency_{latches}$$

This high-level analysis shows that for wider-issue pipelines, the latency of the time compaction increasingly dominates. Therefore, to support these pipelines, low-latency temporal compactors must be investigated.

4.4 Hash Structures

This section introduces the logical structure of the spatial and temporal compactors and their analytic aliasing properties.

4.4.1 Spatial Compactors

This section considers two spatial compactors: an interleaved parity tree and X-Compact-based tree [69]. The spatial compactors reduce sixty-four bits of input data to sixteen bits of output through a tree of carefully arranged XOR gates and preserve all class 1 and most class 2 errors. Because spatial compactors operate on individual instruction results, class 3 errors are not relevant to spatial compaction.

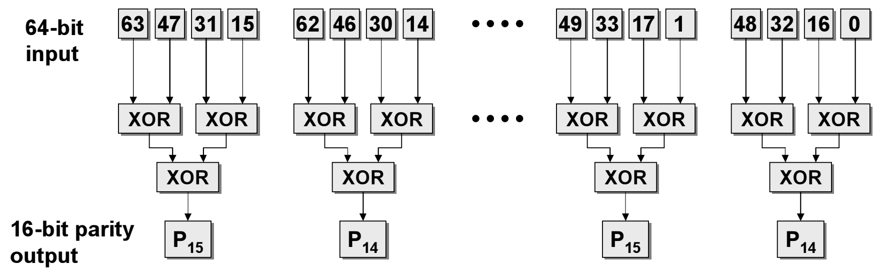


Figure 13. Diagram of a sixty-four to sixteen parity tree-based spatial compactor.

Parity tree. The interleaved parity tree is shown in Figure 13. A parity tree simply computes the XOR of groups of input bits directly into an output bit. In a parity tree, each input bit is represented in exactly one output bit. This instantiation of a parity tree XORs every sixteenth input bit. The interleaved arrangement provides the same statistical aliasing protection as a parity tree that XORs four contiguous input bits together, however the interleaved parity tree specifically provides protection against a common error pattern where an even number of contiguous bits are simultaneously flipped. Such error patterns can be expected with single-event upsets that have a spatial correlation, such as multi-bit upsets from radiation strikes [8, 61].

The interleaved parity tree preserves all single-bit and odd-bit errors. The tree also preserves all contiguous bit errors, except those affecting exactly thirty-two bits. However, it cannot detect all bits flipped and has notably poor performance with smaller even-bit errors, particularly two-bit errors.

Implementation is inexpensive, as the circuit can be built as a balanced, shallow tree of $R \cdot \log N$ XOR gates, where R is the ratio of the outputs to inputs (always less than one). For sixty-four to sixteen bits, this requires three two-input XOR gates per output, arranged two levels deep.

X-Compact tree. The X-compact parity tree [69] is based on error correcting codes and improves preservation of even-bit errors. The tree is described mathematically by an MxN generator matrix which maps a multiplication in GF(2) of M input bits by the matrix to N output bits. Logically, each one in the matrix represents an XOR operation of an input (each row) to produce an output (each column). An example eight-bit input, five-bit output X-compact tree is shown in Figure 14, along with its generator matrix. As proposed, the X-compact parity tree is also known as an *odd-weight*

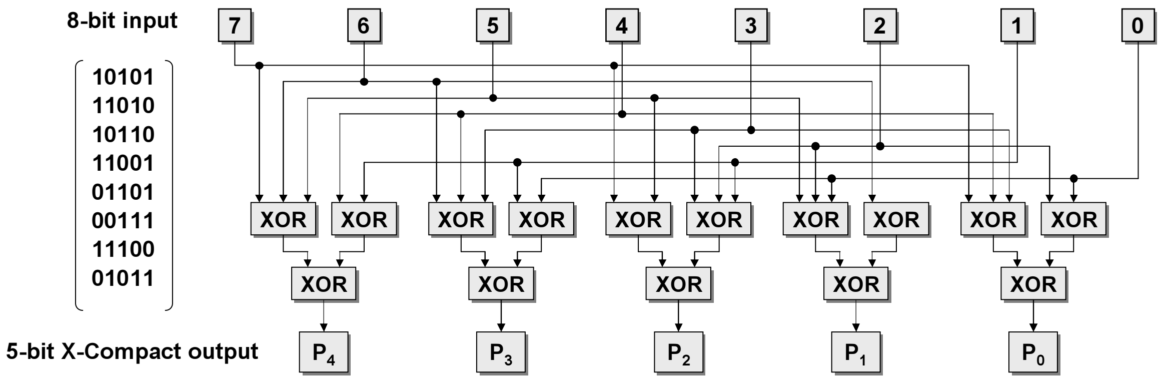


Figure 14: Diagram of an eight-to-five bit X-compact spatial compactor and its corresponding generator matrix.

column code [79].[2] The output is wider than the minimal parity tree configuration, but sixteen bits is still sufficient to preserve the important aliasing properties of the structure. Unlike the parity tree, each input is represented in at least two outputs; furthermore, no output contains the same pair of inputs as another output. These properties guarantee that no pair of bits can cancel each other in the output, and hence provides detection of two-bit errors. The necessary condition comes from error-correcting code theory [69, 79]:

Every row of the generator matrix must be non-zero, distinct, and have odd weight.

The X-compact matrix preserves all single-bit, double-bit, and odd-bit errors, as well as contiguous bit errors of any size. Later in this section, the definition is extended to provide $\approx 2^{-(p-1)}$ aliasing for even-bit errors in a p-bit output compactor.

Implementation is more expensive than a parity tree because of additional levels of XORs. Furthermore, for a given number of inputs and outputs, the possible generator matrix is not unique, in general (for example, the eight-to-five matrix in this figure differs from the eight-to-five matrix in Mitra's X-compact design but has equivalent detection capabilities [69]). Furthermore, as observable in Figure 14, not all paths in the tree are balanced: output bit P_1 has four input bits, while other outputs have five input bits). Therefore, the X-Compact tree is not balanced.

In the next section, this thesis investigates an X-Compact tree with sixty-four inputs and sixteen output bits. In this tree, the outputs each combine between eighteen and twenty-one input terms for each output and the weight of each generator matrix row is five. This tree uses five levels of two-

[2]In information theory, the term *weight* refers to the number of ones in a binary number.

input XOR gates. As a point of comparison, the single error detect, double error detect (SEC/DED) ECC logic used in the OpenSPARC T1's computes an eight-bit syndrome for a sixty-four bit word, with a range of eight to thirty-five inputs per output bit. That tree requires an additional level of XOR gates.

The selected X-Compact tree is also not minimal, given a 64-bit SEC/DED ECC code needs only eight syndrome bits and a similar X-compact tree needs nine bits (an X-compact matrix also guarantees detection of one and two-bit errors in the presence of one unknown value, although this property is not needed for in this thesis). Instead, the compactor needs sixteen bits of output to feed the temporal compactor and maintain the desired level of aliasing.

Simply making the generator matrix rows wider, while keeping the weight constant does not guarantee an improved level of aliasing (a trivial example where this is true is by adding a column of zeros). To build this compactor, the necessary conditions are extended with the following condition:

Every column of the generator matrix must be non-zero and distinct.

This condition is implicitly satisfied for minimal-sized outputs using the prior conditions. However, by requiring the columns to be non-zero, this ensures that the output detects some errors (otherwise, the output yields no new information). By requiring the columns to be distinct, each additional column adds further detection capabilities. A duplicate column means two outputs are always be the same, giving no new detection for the additional output, while the distinction requirement provides proportionally stronger detection of otherwise-aliased even-bit errors, reaching $\approx 2^{-(p-1)}$ for even-bit errors in a compactor with p outputs.

4.4.2 Temporal Compactors

This section considers four temporal compactors: XOR-and-rotate, checksum, multiple-input shift register (MISR), and cyclic redundancy check (CRC). The compactors preserve all class 1, virtually all class 2, and varying degrees of class 3 errors. Figure 15 illustrates the combinational logic form of each compactor and uses the following notation: I_n for the n input bits from a spatial compactor, P_n for the n input bits from a previous stage (either registers storing a hash from a previous cycle or a combinational stage from another temporal compactor), and O_n for the n output bits.

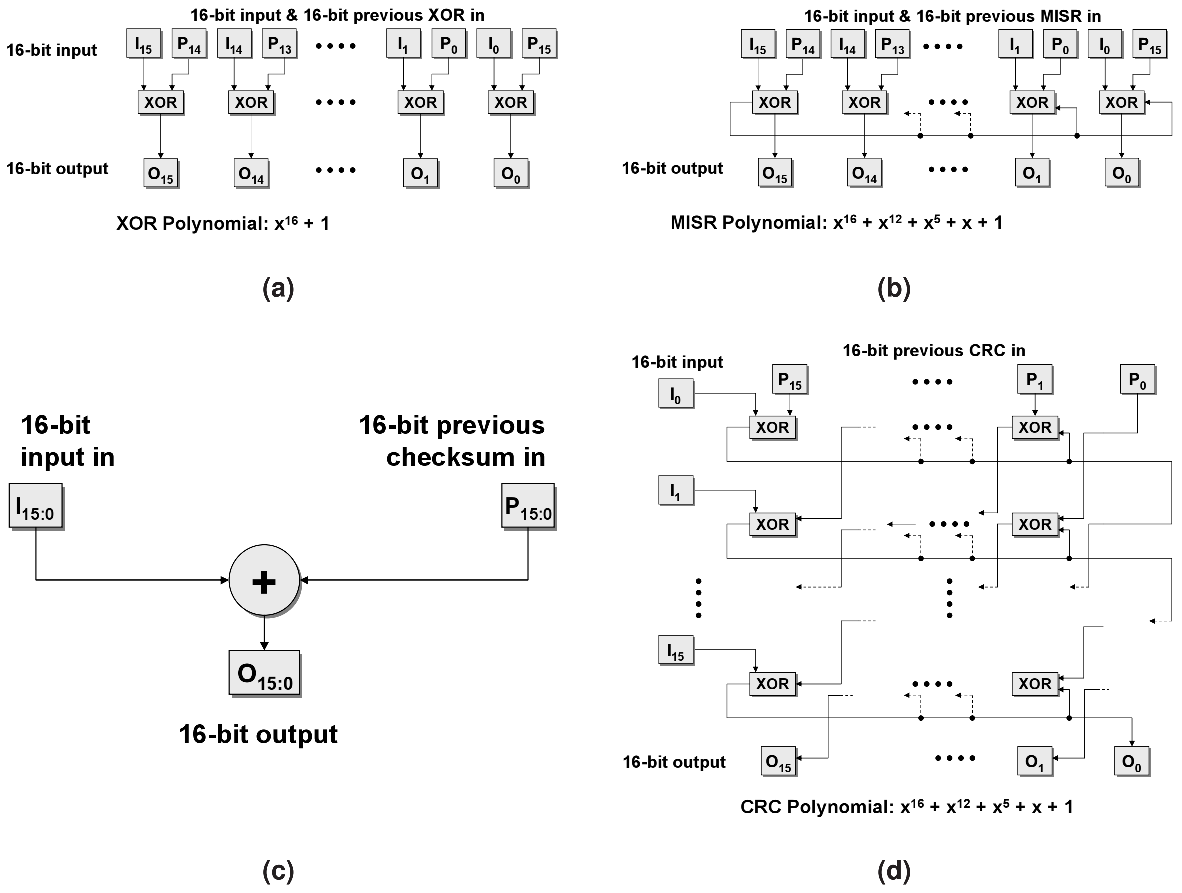

16-bit input & 16-bit previous XOR in
16-bit input
XOR
16-bit output
XOR Polynomial: x^16 + 1
(a)
16-bit input & 16-bit previous MISR in
MISR Polynomial: x^16 + x^12 + x^5 + x + 1
(b)
16-bit input in
16-bit previous checksum in
16-bit output
(c)
16-bit input
16-bit previous CRC in
CRC Polynomial: x^16 + x^12 + x^5 + x + 1
(d)

Figure 15. Diagram of temporal compactors.

XOR-and-rotate. The XOR-and-rotate unit is illustrated in Figure 15(a). Mathematically, this unit can be thought of as a shift register with a generator polynomial of $x^{16} + 1$.

Because it is based on a simple XOR function for each bit, with a wraparound for the shift out, the class 2 error pattern of all ones is always aliased in this compactor. All other class 1 and 2 errors are preserved. Although the class 3 aliasing rate does eventually converge to $2^{-(p-1)}$, this is only true when most instructions in the interval contain errors. Performance is particularly poor if only a few instructions are in error.

The cost of implementation is simply a single XOR gate for each input, plus wiring to implement the rotation. This compactor is the least expensive of the ones considered and requires only a single level of XOR gates.

MISR. The multiple-input shift register (MISR) has a similar structure to the XOR-and-rotate compactor, as illustrated in Figure 15(b). Unlike the previous compactor, this compactor uses the well-known CCITT 16-bit primitive polynomial to compute the next hash. Other primitive polynomials give comparable performance for a given MISR width.

Unlike XOR-and-rotate, the MISR preserves all class 2 errors. This situation is rectified because in the all one's error case, while the wraparound bit cancels the least-significant input bit, the remaining terms in the generator polynomial are also inverted. Hence, the error is preserved in the MISR. For class 3 errors, the MISR shows weaknesses similar to XOR-and-rotate, particularly with small numbers of bit errors, however it quickly converges to $2^{-(p-1)}$. This weakness occurs because an error in one cycle can exactly cancel with a (rotated) error pattern in a subsequent cycle's input, before the previous error has been "distributed" throughout the hash by the wraparound in the most-significant bit.

The implementation cost is similar to the XOR-and-rotate, but with an additional cost of an XOR-gate for each term in the generator polynomial. As such, it can be implemented in two levels of two-input XOR gates.

Checksum. The checksum unit consists of a combinational adder which sums the spatial input and the previous checksum to create a new checksum, as illustrated in Figure 15(c). The adder's carry out bit can be stored as input to a later checksum or discarded.

The implementation that ignores the carry out bit can detect all class 2 errors, which can be algebraically verified by adding any single error pattern to the input and showing that the subsequent output always differs by an amount equal to the error pattern's value. By contrast, the implementation with a carry out aliases when the error-free input is zero and the error pattern is all ones. Class 3 errors for both forms of the checksum show performance similar to the MISR for two instances of burst errors over a fingerprint interval, but converge to 2^{-p} average aliasing rate with further errors.

CRC. The cyclic redundancy check (CRC) unit is a combinational realization of the familiar linear feedback shift register (LFSR), unrolled over for N steps, one for each input bit, as illustrated in Figure 15(d). As with the MISR, this study uses the CCITT-16 polynomial as a basis for the CRC.

The CRC preserves all class 1 and 2 errors. Furthermore, because every error bit reaches the high-order bit position and is spread throughout the hash before subsequent inputs are added, the CRC has strong and consistent properties for class 3 errors, which are uniformly $2^{-(p-1)}$.

The CRC does not come without cost, however. To the first order, the CRC costs roughly N times as much as an equivalent N-bit MISR with the same primitive polynomial. The comparable logic depth is roughly thirty-two two-input XOR gates for the CRC-16 and is difficult to optimize, which makes it the most expensive of the hashes both in terms of area and latency.

4.5 Evaluation

This section empirically evaluates the aliasing properties of the spatial and temporal compactors. First, the compactors are considered separately, then the combination of spatial and temporal compactors are evaluated together. This section concludes with a synthesis-based evaluation of the full compactor area and latency properties for scalar and four-wide superscalar pipelines.

4.5.1 Methodology

The compactors in this thesis are evaluated using a C-language program that models the retirement stage of a pipeline and the architectural fingerprint compactor. The retirement stage is assumed to retire uniform random values to its architectural registers. Both golden (error-free) and error-injected inputs are fed to the compactor model and their results are compared after a chosen fingerprint interval.

Architectural errors are modeled as bit flips uniformly spread across the input bits. The number of bit flips in the output is specified for each run. The program injects error of a specified size (e.g., number of bit flips) uniformly across input words. For example, a two-bit error has $\begin{pmatrix} n \\ 2 \end{pmatrix}$ possible error patterns over an n-bit input. Because the crossproduct of the possible inputs and error patterns is enormous, 10^8 trials are run for each combination of compactor and number of input bits in error. While this is only a sampling of the possible error space, 10^8 trials yields results within $\pm 5\%$ with 95% confidence for all results. Note that most other sampling experiments in this thesis can achieve similar levels of statistical confidence with orders of magnitude fewer trials, however as the probability of aliasing approaches zero, the number of trials increases dramatically.

For synthesis results, each compactor is modeled in Verilog with a single sixteen-bit register to store the result. For four-wide superscalar pipelines, a priority-encoded mux is also included to

Table 5: Aliasing properties for spatial compactors with uniform random bit errors over a 64-bit word. For reference, $2^{-p-1} \approx 0.00003052$ for p = 16.

	1-bit	2-bit	Odd-bit	Even-bit > 4	All bits
Interleaved Parity Tree	0	0.0476	0	$<\approx 2^{-(p-1)}$	1
X-Compact Tree	0	0	0	$2^{-(p-1)}$	0

select between the four possible instruction outputs, depending on the state of the retirement valid signals. The synthesis methodology is the same as described in Section 4.3.2.

4.5.2 Empirical Aliasing Properties

Aliasing in spatial compactors. First, this section evaluates the aliasing properties of the 64-to-16 bit spatial compactors. Table 5 shows the probability of aliasing for the interleaved parity tree and X-Compaction tree for error classes 1 (first column) and 2 (remaining columns) over a 64-bit word. Because spatial compactors have no memory of previous instructions, class 3 errors are not relevant.

As expected, both compactors preserve all class 1 errors. However, the interleaved parity tree shows poor performance with two-bit errors, aliasing nearly 5%. Aliasing occurs on any pattern where two bits in an interleaved group of four are in error. Larger even-bit errors range from that value to $\approx 2^{-(p-1)}$ (with four and sixty-two-bit errors being the worst). By contrast, the X-Compact's performance for all sizes of even-bit errors is consistently at $\approx 2^{-(p-1)}$. For those readers who are now reaching for a calculator, $2^{-15} \approx 0.00003052$ and $2^{-16} \approx 0.00001526$. Note that the even-bit errors cover half the possible input errors; accounting for the odd-bit errors—which are always preserved—still yields the overall aliasing limit of 2^{-p} for p output bits in both compactors.

Errors in output. Next, this section investigates the number of output bits that differ with respect to an error-free output (termed *output bits in error*). The purpose of this result will be clear only after presenting the aliasing properties of the temporal compactors.

Figure 16 shows the probability density function for the average number of output bits in error over the entire space of errors, consisting of equally-weighted instances of single-bit up to sixty-

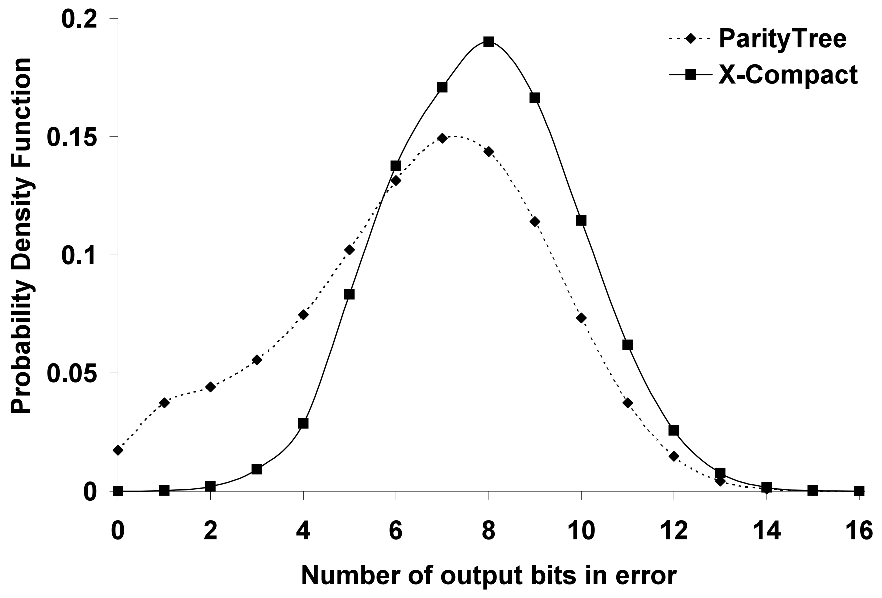


Figure 16: PDF of the number of bit errors generated by the spatial compactors for a uniform random incidence and placement of bit errors.

four bit errors for each compactor. The left extreme left point (zero bits in error), shows the overall aliasing probability (0.017 and 2^{-p} for parity tree and X-Compact, respectively). The X-Compact curve centers around eight bits in error. By contrast, the parity tree is heavily biased towards smaller numbers of bit errors in its output, because each input error leads to exactly one output. As a digression, the X-Compact curve roughly follows, but does not perfectly fit a Gaussian distribution with a mean of eight and sigma of 2.1. This is because 64-bit errors always produce a six-bit output error in this tree, which induces a small bias towards six.

The take-away message from Figure 16 is that the X-Compact tree amplifies all error classes into many-bit errors, while the parity tree hash exhibits higher aliasing and produces more few-bit errors for the same input patterns. This property will be useful for temporal compactors that are strong in preserving many-bit errors and relatively poor at preserving few-bit errors.

Another way to view this data is by the average number of bit errors propagated to the output as a function of the bit errors in the input. Ideally, for uniform random inputs, the outputs should also demonstrate uniform random outputs—on average half in error and half correct—which yields eight bits for these compactors. The actual performance of the compactors is shown in Figure 17. This figure shows that for single-bit errors (extreme left) the parity tree produces a single-bit error in output while the X-Compact tree produces exactly five bits in error. These values, unsurprisingly,

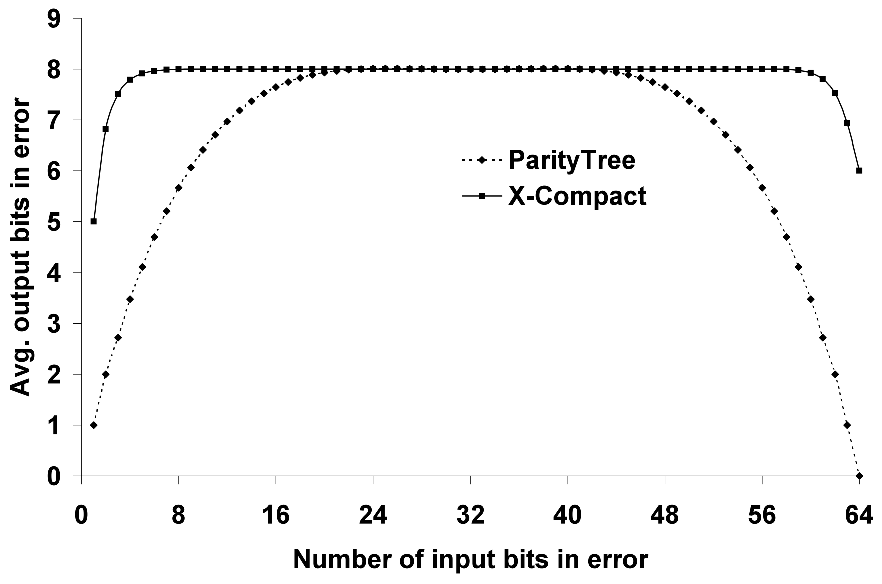


Figure 17: The number of bits propagated in error output as a function of the number of bits in error in the input for spatial compactors.

match the weight of the generator matrices for each compactor—that is, each input bit is connected to exactly one and five outputs, respectively. More interestingly, the X-Compact tree quickly adds and maintains more output bits in error with additional errors in the input. By contrast, the parity tree retains fewer average output bits in error than the X-Compact tree in the ranges of $(0, 16]$ and $[48, 64]$ input bits in error. Therefore, the parity tree is weaker than the X-Compact tree at preserving many-bit errors in the output. Finally, the plateau at eight bits is expected, because the sixteen output bits are essentially uniform random and thus each bit has a $\frac{1}{2}$ probability of aliasing.

Aliasing in Temporal compactors. This section now evaluates the aliasing properties of the temporal compactors. Class 1 and 2 errors were described analytically in the previous section and are therefore their properties are not repeated here. Instead, this section concentrates on class 3 errors over different fingerprint intervals. Because the compactors have different behaviors for class 3 errors, depending on how many errors have propagated in time, the extreme cases are examined first to establish bounds on the behaviors. In all cases, fingerprint intervals of ten instructions are studied, because the aliasing results showed no change with larger intervals (e.g., 100-1,000 instructions). Instruction words and bits are selected for error injection randomly, with a uniform distribution across the fingerprint interval.

Table 6: Aliasing properties for temporal compactors with uniform random bit errors over two 16-bit words. For reference, $2^{-p-1} \approx 0.00003052$ for p = 16.

	1-bit	2-bit	Odd-bit	Even-bit > 4	All bits
XOR	0.0625	0.0084	0.0019	0.0038	1
Checksum	0.0332	0.0026	$\approx 2^{-(p-1)}$	$\approx 2^{-(p-1)}$	$\approx 2^{-(p-1)}$
MISR-CCITT	0.0482	0.0054	$\approx 2^{-(p-1)}$	$\approx 2^{-(p-1)}$	0
CRC-CCITT	0	$\approx 2^{-(p-1)}$	$\approx 2^{-(p-1)}$	$\approx 2^{-(p-1)}$	0

Table 7: Aliasing properties for temporal compactors with uniform random bit errors over ten 16-bit words. For reference, $2^{-(p-1)} \approx 0.00003052$ and $2^{-p} \approx 0.00001526$ for p = 16.

	1-bit	2-bit	Odd-bit	Even-bit > 4	All bits
XOR	0.0006	$\approx 2^{-(p-1)}$	$\approx 2^{-(p-1)}$	$\approx 2^{-(p-1)}$	1
Checksum	0.0001	$\approx 2^{-p}$	$\approx 2^{-p}$	$\approx 2^{-p}$	$\approx 2^{-(p-1)}$
MISR-CCITT	0.0003	$\approx 2^{-(p-1)}$	$\approx 2^{-(p-1)}$	$\approx 2^{-(p-1)}$	0
CRC-CCITT	$\approx 2^{-(p-1)}$	$\approx 2^{-(p-1)}$	$\approx 2^{-(p-1)}$	$\approx 2^{-(p-1)}$	0

First, consider the case where precisely two instructions in the interval have errors. Table 6 shows the aliasing probability for two instances of an N-bit error over the interval. For two instances of single-bit errors, the XOR, checksum, and MISR all show poor aliasing performance. In these designs, a single input bit error can produce a single bit error in the output. A subsequent single-bit error in a following instruction can then cancel the original error, thus causing aliasing. The checksum and MISR circuits perform better than the XOR because the initial error can also cause multiple bit flips (through a carry, or by being spread after reaching the most significant bit position, respectively), which explains their better performance relative to the XOR. The CRC preserves all two-bit errors if the total number of input bits is less than $2^{(p-1)}$ [84] (512 instructions).

For two-bit and larger errors, the likelihood of a subsequent error exactly canceling the initial error is lower (because the probability of matching the first error pattern decreases combinatorially). For all bits in error, the XOR compactor exactly cancels the error patterns, while the MISR and CRC always detect these burst patterns.

The result from Table 6 shows a weakness among the XOR, checksum, and MISR for few-bit errors. This observation motivates the need for spatial compactors that generate many-bit differences in their output.

Table 8: Overall Aliasing properties. For reference, $2^{-(p-1)} \approx 0.00003052$ and $2^{-p} \approx 0.00001526$ for p = 16.

Compactor	**1-bit**	**2-bit**	**Odd-bit**	**Even-bit > 4**	**All bits**
			Class 1/2 errors		
Parity Tree - *	0	0.0476	0	0.0022	1
X-Compact - *	0	0	0	$\approx 2^{-(p-1)}$	0
			Class 3 – Two instructions in error		
Parity Tree - XOR	0.0625	0.0098	0.0002	0.0004	1
Parity Tree - Checksum	0.0332	0.0046	0.0001	0.0002	1
Parity Tree - MISR	0.0482	0.0072	0.0002	0.0003	1
Parity Tree - CRC	0	0.0024	$\approx 2^{-(p-1)}$	0.0001	1
X-Compact - XOR	0.0028	0.0001	$\approx 2^{-(p-1)}$	$\approx 2^{-(p-1)}$	0.1777
X-Compact - Checksum	0.0006	$\approx 2^{-(p-1)}$	$\approx 2^{-p}$	$\approx 2^{-p}$	0.0156
X-Compact - MISR	0.0001	$\approx 2^{-(p-1)}$	$\approx 2^{-(p-1)}$	$\approx 2^{-(p-1)}$	0
X-Compact - CRC	0	$\approx 2^{-(p-1)}$	$\approx 2^{-(p-1)}$	$\approx 2^{-(p-1)}$	0
			Class 3 – All instructions in error		
Parity Tree - XOR	0.0006	0.0001	$\approx 2^{-(p-1)}$	$\approx 2^{-(p-1)}$	1
Parity Tree - Checksum	0.0001	$\approx 2^{-p}$	$\approx 2^{-p}$	$\approx 2^{-p}$	1
Parity Tree - MISR	0.0003	$\approx 2^{-(p-1)}$	$\approx 2^{-(p-1)}$	$\approx 2^{-(p-1)}$	1
Parity Tree - CRC	$\approx 2^{-(p-1)}$	$\approx 2^{-(p-1)}$	$\approx 2^{-(p-1)}$	$\approx 2^{-(p-1)}$	1
X-Compact - XOR	$\approx 2^{-(p-1)}$	$\approx 2^{-(p-1)}$	$\approx 2^{-(p-1)}$	$\approx 2^{-(p-1)}$	0
X-Compact - Checksum	$\approx 2^{-p}$	$\approx 2^{-p}$	$\approx 2^{-p}$	$\approx 2^{-p}$	0.0038
X-Compact - MISR	$\approx 2^{-(p-1)}$	$\approx 2^{-(p-1)}$	$\approx 2^{-(p-1)}$	$\approx 2^{-(p-1)}$	0
X-Compact - CRC	$\approx 2^{-(p-1)}$	$\approx 2^{-(p-1)}$	$\approx 2^{-(p-1)}$	$\approx 2^{-(p-1)}$	0

Next, this section studies the other extreme case where all instructions in the interval experience an N-bit error. Table 7 shows the aliasing properties for the four compactors. As with two instruction errors, the same compactors show relatively weak aliasing performance for single-bit errors. However, the likelihood of aliasing drops to the theoretical minimums with larger-scale error patterns. This result is consistent with the results in the prior table and again indicates that many-bit errors are better protected (with the exception of XOR). Results between the two extreme cases converge quickly (within 3% for longer fingerprint intervals) of the values in Table 7 and are omitted from this document because little additional information can be learned from them.

Combined Spatial-Temporal Compactor Aliasing. This section now evaluates the overall aliasing properties of compactors with the combined spatial and temporal compactors. Errors are injected into the spatial compactor inputs and the temporal compactor's final output. The final hash is compared with the equivalent error-free hash.

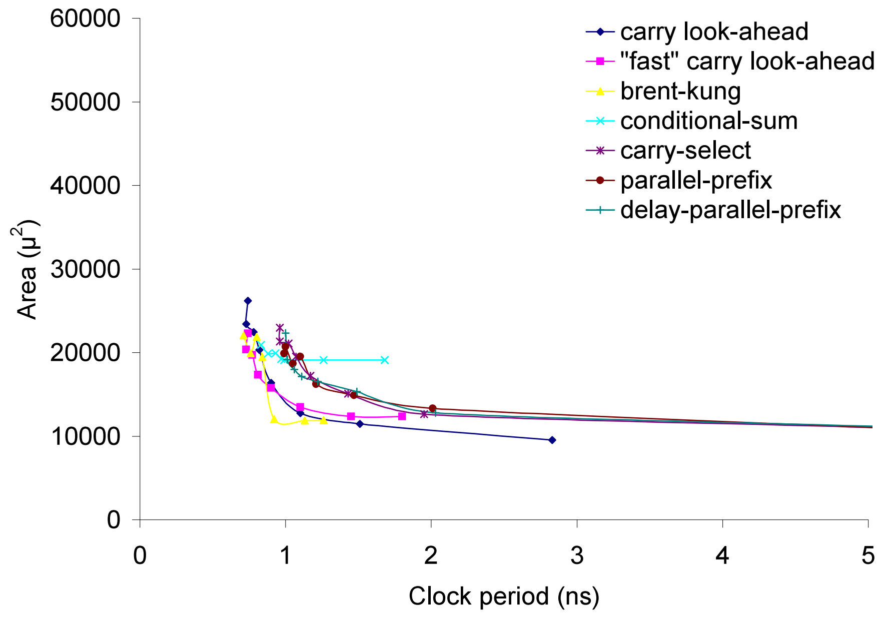


Figure 18: The area-latency curves for a range of reference sixty-four bit adder implementations.

Table 8 summarizes the results. For class 1 and 2 errors, the aliasing properties are determined entirely by the spatial compactor. This is true because, as discussed earlier, the temporal compactors have strong preservation properties for class 1 and 2 errors. As shown in the table, the X-Compact-based units show significantly better performance than the parity trees for even-bit errors.

For class 3 errors, the results are again divided by the number of instruction words in error within the fingerprint interval. First, for two instruction words in error, the X-Compact-based compactors are consistently better than the parity tree-based compactors—by orders of magnitude for single and double-bit errors. In addition, the spatial compactor's skew towards many-bit errors helps the inexpensive MISR-based compactor achieve average aliasing performance nearly identical to the CRC. Similar results are evident in the last section of the table, where all instruction words contain errors. Here, the X-Compact-based compactors are also clearly better. Furthermore, all the temporal compactors have strong aliasing performance, with the checksum having half the aliasing of the others.

4.5.3 Synthesis Results for latency and area

This section now studies the area-latency tradeoffs for the different compactors. Figure 18 shows a reference plot for several different sixty-four bit adders. The adder serves as a useful reference

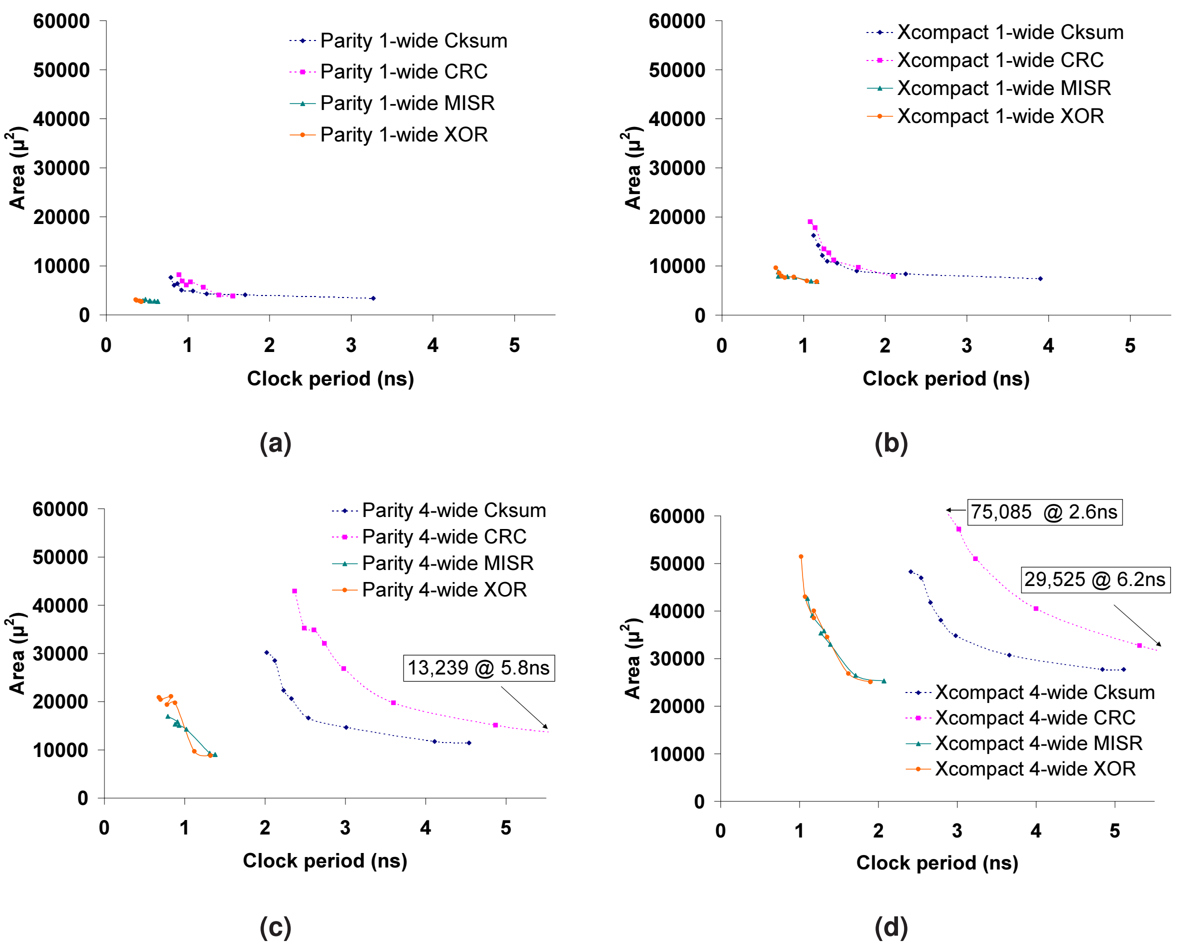


Figure 19: The area-latency curves for (a) a scalar pipeline with parity tree compaction, (b) scalar pipeline with X-Compact compaction, (c) four-wide superscalar pipeline with parity tree compaction, and (d) four-wide superscalar pipeline with X-Compact compaction.

because it matches the width of the pipeline that architectural fingerprints protect. Note that Design Compiler is instructed to choose a particular adder, but then maps to the ASIC library and applies further optimizations that can change the final adder design. For comparison purposes with the remainder of this section, the best adder latency is roughly 0.7ns (carrying a corresponding area of 22,050 μ^2), while the best area is 9,556 μ^2. In the checksum designs, Design Compiler chooses a sixteen-bit carry-lookahead adder implementation as the basis for synthesis.

Figure 19 shows a Pareto-style curve for area versus latency for each of the compactor combinations in both scalar and four-wide superscalar configurations. Comparing the graphs horizontally, the X-Compact incurs roughly a double area overhead compared to parity trees, although the best latency values are still comparable. As expected, the XOR and MISR designs closely follow each other in all area-latency tradeoffs because their implementation differs only by a handful of XOR gates. Based upon the aliasing evidence, this indicates that the MISR is a clear win for virtual no additional area or latency costs.

The checksum and CRC tend to have similar area-latency tradeoffs, although both implementations are consistently larger and slower than the XOR and MISR designs. As predicted analytically, the area costs scale roughly in proportion to the pipeline width. That is, the superscalar pipeline units are consistently four times the area and latency of the scalar pipeline units.

The X-Compact/MISR scalar pipeline implementation is half the size of the reference adder and has equal latency. Hence, this hash design is affordable for scalar pipeline implementation. The CRC and checksum-based units are roughly the same size as, but significantly slower than, the reference adder.

For the superscalar pipeline, the area and latency differences among the compactor units are more apparent. The X-Compact/MISR unit is roughly twice the cost of the reference adder (the comparable pipeline is more than four times the area and complexity of a scalar pipeline, therefore the relative overhead is lower). Latency scales with the latency of the time compactor, which fortunately, is small. Therefore, the best compactor latency is 45% longer than the best adder latency; however, a pipeline between the spatial and temporal compactors can bridge this gap. By contrast, the CRC-based compactors are nearly four times the size of the adder and over three times the latency. This gap cannot be recovered by pipelining.

The take-away result from this evaluation is that the MISR-based temporal compactors provide the best area-latency tradeoffs. Furthermore, the X-Compact/MISR combination has comparable aliasing properties to a CRC, however with several times lower area and latency.

Pipelined Compactors. Next, pipelined compactors are investigated using synthesis. A single pipeline stage is added between the spatial compactors and the temporal compactors. This stage maintains the same overall throughput as the original compactor; however, the required cycle time decreases. These latches increase area marginally, while the minimum clock period decreases. The pipeline introduces no change in the aliasing properties of the compactor.

The area-latency tradeoff with a pipeline stage is presented in Figure 20. The parity tree results show little change in latency from the added pipeline stage, because the parity tree is already shallow and its latency roughly equals the latency overhead of the new registers. By contrast, the X-Compact

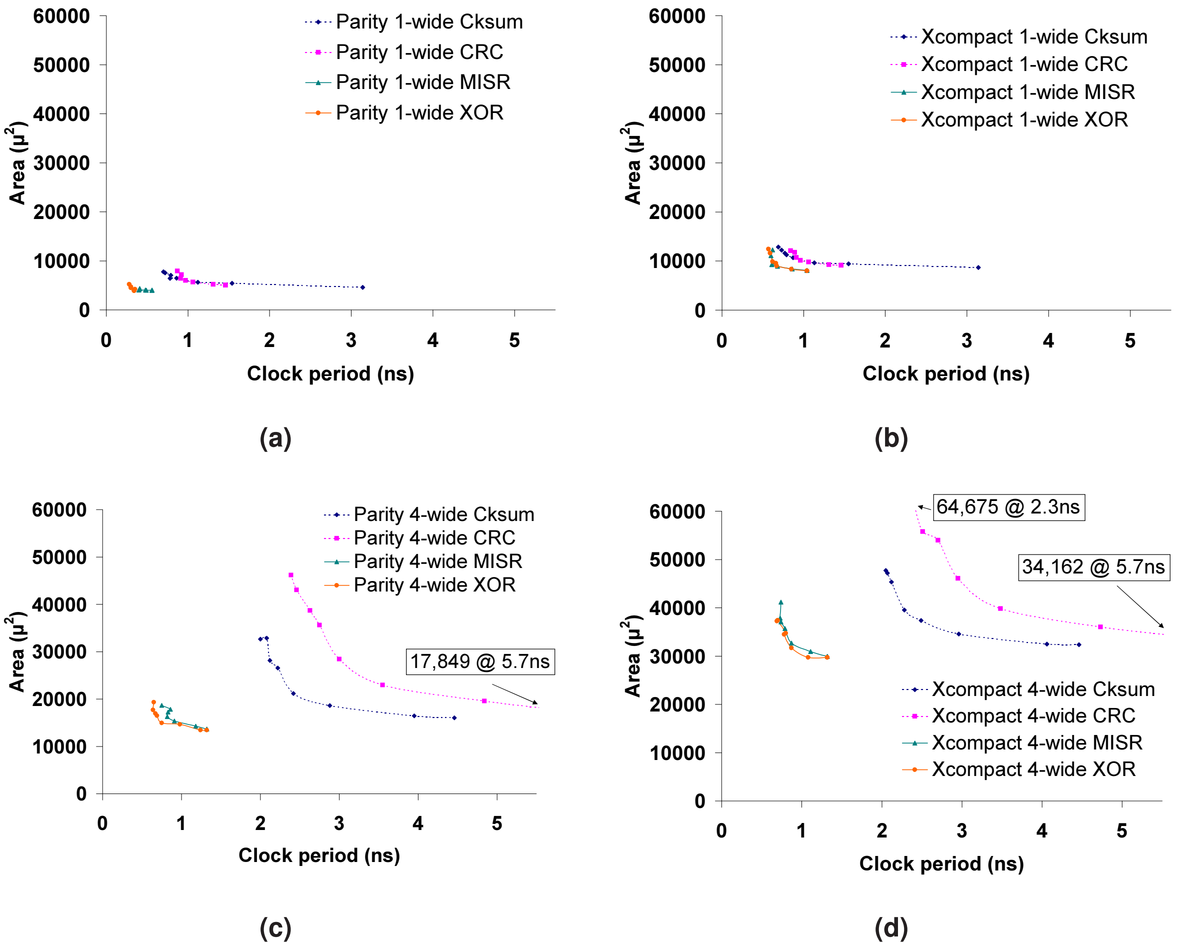


Figure 20: The area-latency curves for pipelined compactors with (a) a scalar pipeline with parity tree compaction, (b) scalar pipeline with X-Compact compaction, (c) four-wide superscalar pipeline with parity tree compaction, and (d) four-wide superscalar pipeline with X-Compact compaction.

tree is deep enough to provide some benefit. Surprisingly, the scalar version has both better latency and marginally lower area for the checksum and CRC designs than without a pipeline. The area is roughly half a reference adder and the latency matches the adder. The most encouraging gains are in the superscalar case, where the XOR and MISR-based compactors match the latency of the adder, still with twice the area cost.

4.6 Conclusion

This chapter investigates the implementation of hash circuits for architectural fingerprints along error coverage, area, and latency axes. A scalable hash architecture is presented that allows latency and area-efficient implementations. The ASIC synthesis and error injection simulation results from this study show that a hash architecture using a combination of ECC-like X-Compact trees and MISRs, commonly applied to scan chain compaction in the manufacturing test domain, can accept the architectural state retirement bandwidth of modern wide-issue superscalar processors, while maintaining the near-ideal error coverage of a CRC.

Chapter 5

Reunion

5.1 Introduction

Chip multiprocessors (CMPs) have emerged as a promising approach to give computer architects scalable performance and reasonable power consumption within a single chip [15, 64, 77]. However, increasing levels of integration, diminishing node capacitance, and reduced noise margins have led researchers to forecast an exponential increase in the soft-error rate for unprotected logic and latch circuits [48, 97]. Recent work [40, 72, 114] advocates leveraging the inherent replication of processor cores in a CMP for soft-error tolerant redundant execution by pairing cores and checking their execution results.

Because CMP designs maintain the familiar shared-memory programming model, multicore redundant architectures must provide correct and efficient execution of multithreaded programs and operating systems. Furthermore, redundant execution must not introduce significant complexity over a non-redundant design. Ideally, a single design can provide a dual-use capability by supporting both redundant and non-redundant execution.

Redundant designs must solve two key problems: maintaining identical instruction streams and detecting divergent execution. Mainframes, which have provided fault tolerance for decades, solve these problems by tightly lockstepping two executions [16, 101]. Lockstep ensures both processors observe identical load values, cache invalidations, and external interrupts. While conceptually simple, lockstep becomes an increasing burden as device scaling continues[18, 66].

Researchers have proposed several alternatives to lockstep within the context of CMPs. Both Mukherjee et al. [72] and Gomaa et al. [40] use a custom load-value queue (LVQ) to guarantee that redundant executions always see an identical view of memory. A leading core directly issues loads to the memory system, while a trailing core consumes a record of load values from the LVQ. Although the LVQ produces an identical view of memory for both executions, integrating this strict input replication into an out-of-order core requires significant changes to existing highly optimized microarchitectures [72].

Strict input replication forbids using existing cache hierarchies for the redundant execution and requires changes to critical components of the processor core and cache hierarchy. In contrast, relaxed input replication permits redundant executions to issue independently memory operations to existing cache hierarchies. This thesis observes that, even for shared-memory parallel programs, relaxed input replication produces the correct result in virtually all cases. In the case when load values differ between the redundant cores, called input incoherence, mechanisms for soft error detection and recovery can correct the difference [88].

This thesis proposes the Reunion execution model, which exploits relaxed input replication for soft-error tolerant redundant execution across cores. While Reunion allows redundant cores to issue memory operations independently, Reunion designs are proven in this chapter to maintain correct execution with existing coherence protocols and memory consistency models. Reunion provides detection and recovery from input incoherence using a combination of architectural fingerprints and the existing precise exception rollback—the same mechanisms needed for soft-error tolerance.

The following contributions are made:

- **Input incoherence detection.** This thesis observes that light-weight detection mechanisms for soft errors can also detect input incoherence. This observation enables a single recovery strategy for both soft errors and input incoherence.

- **Reunion execution model.** This thesis presents formal requirements for correct redundant execution using relaxed input replication in a multiprocessor. These requirements do not change the existing coherence protocol or memory consistency model.

- **Serializing check overhead.** This thesis observes that checking execution at instruction retirement incurs stalls on serializing events, such as traps, memory barriers, and non-idempotent instructions. Architectures that encounter frequent serializing events will suffer a substantial performance loss with any checking microarchitecture.

This thesis evaluates Reunion in a cycle-accurate full-system CMP simulator. The Reunion execution model is demonstrated to have an average 9% and 8% performance impact on commercial and scientific workloads, respectively, with a 5-6% performance overhead from relaxed input replication.

Chapter Outline. Section 5.2 presents background on soft error detection and redundant execution. Section 5.3 presents the Reunion execution model, while a CMP implementation is discussed in Section 5.4 and its performance is evaluated in Section 5.5. This chapter concludes in Section 5.6.

5.2 Background

This section covers background topics on soft error tolerant microarchitectures. The section first introduces the fault model assumed for the Reunion Execution Model, then provides a brief overview of existing work in this space. The section also introduces the fundamental requirements for redundant execution and the concept of input incoherence.

5.2.1 Fault Model

The fault model targets soft errors that cause silent data corruption, such as transient bit flips from cosmic rays or alpha particles. This work assumes that the processor's datapath is vulnerable to soft errors from fetch to retirement, but that the less-vulnerable control logic [97] is protected by circuit-level techniques. Designers already protect cache arrays and critical communication buses with information redundancy (e.g., ECC) [101]. However, the complex layout and timing-critical nature of high-performance processor datapaths precludes these codes within the pipeline.

This thesis investigates microarchitectures that detect and recover from virtually all soft errors, but in very infrequent cases, can leave them undetected or uncorrected. Architects design microprocessors to meet soft error budgets [71] and this design can be engineered to meet the desired budget.

Array protection. Unretired speculative pipeline state, such as a speculative physical register file and the issue queue, can remain unprotected. However, protection on retired architectural state arrays, such as the architectural register file and non-speculative store buffer, depends upon the desired soft error protection budget and is independent of the Reunion Execution Model.

The level of protection on architectural arrays can range from completely unprotected to full ECC protection. For example, if the desired soft error budget permits detected, uncorrectable errors in these structures, but silent data corruption is unacceptable, parity protection is acceptable. While this design point is cheaper and easier to implement than ECC, this design point means that "safe state" (as defined in Section 5.3) can contain detectable, but uncorrectable errors. Alternatively, if single-bit errors in the register file must be correctable events in the soft error budget, then the register file must also include SECDED ECC protection.

Protection of the small, performance-critical architectural arrays is not without silicon area and timing costs. Several recent industrial designs have accepted these implementation costs and chose to include parity on the register file [64, 8], store buffer [113], or even ECC on the register file [113] and L1 caches [6, 8].

5.2.2 Redundant Execution

The ”sphere of replication” defines three general design requirements for all systems with redundant execution [88]. First, all computation within the sphere must be replicated. Second, all inputs entering the sphere must be replicated for each execution. Finally, all outputs leaving the sphere must be checked to prevent errors from propagating outside the sphere.

This thesis now discusses the two dominant forms of redundant execution in microprocessors in industry and research communities: lockstep and multithreading.

Lockstep. Classical lockstep redundant execution where identical processing elements are tightly-coupled on a cycle-by-cycle basis has long existed in mainframes such as HP NonStop [16] and IBM zSeries [101]. However, lockstep in general-purpose execution encounters significant roadblocks in future process technologies. First, individual cores are likely to operate in separate clock domains for dynamic frequency control, while execution must still match precisely in time despite asynchronous inputs and physical distances between the cores [18, 66]. Second, increasing within-die

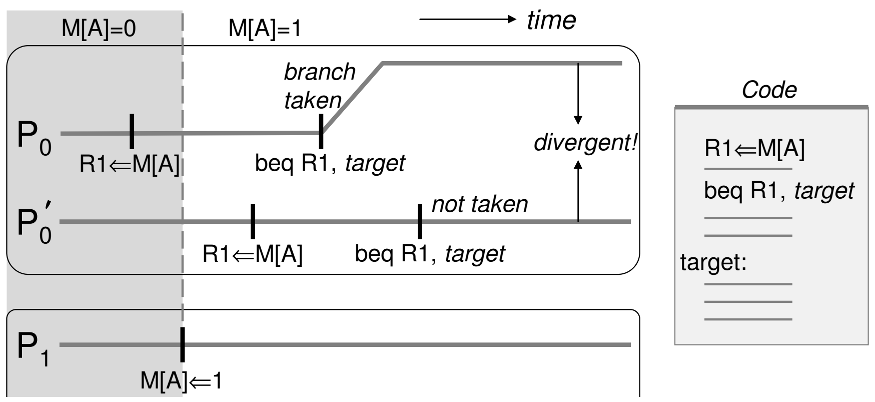


Figure 21: Input incoherence: redundant cores P_0 and P_0' observe different values for memory location $M[A]$ because of an intervening store.

device- and circuit-level variability [22] leads to deviations from precise lockstep because, even in the absence of errors, cores will no longer have identical timing properties or execution resources. Third, lockstep requires precise determinism and identical initialization across all processor components, including in units that do not affect architecturally correct execution (e.g., branch predictors [73]). As a result, redundant execution models that avoid lockstep are highly desirable.

Lockstep meets the sphere of replication requirements by construction. Execution between lockstepped units is redundant, all inputs must be replicated in order to maintain the lockstep, and outputs are trivially compared.

Multithreading. Recent proposals investigate using independent redundant threads within a simultaneous multithreaded (SMT) core [88, 115] or across cores in a CMP [40, 72, 114]. Unlike lockstep, the threads execute independently and the threads are therefore bound by architectural requirements rather than microarchitectural timing constraints. Threads synchronize as outputs from the core (e.g., store values or register updates) are compared, but remain coupled within a short distance to limit the storage needed for input replication and output comparison.

Redundant multithreading permits a range of designs that meet the sphere of replication requirements. These designs are explored in the following text.

5.2.3 Input Incoherence

Multithreading introduces a problem for redundant execution because the threads independently execute and issue redundant memory requests. When executing shared-memory parallel programs, the threads can observe different values for the same dynamic load, which we term input incoherence due to data races. Figure 21 illustrates this situation: these races arise between one execution's read of a cache block and the redundant partner's corresponding read. Writes from competing cores will cause input incoherence. This occurs in ordinary code such as spin-lock routines.

To avoid input incoherence, several prior proposals [114, 72, 88, 115] enforce strict input replication across the redundant threads, where a leading execution defines the load values observed by both executions. Strict input replication can be achieved by either locking cache blocks or recording load values.

The active load address buffer (ALAB) [88] tracks cache blocks loaded by the leading thread and prevents their replacement until the trailing thread retires its corresponding load. The ALAB adds multiported storage arrays to track accessed cache blocks, logic to defer invalidations and replacements, and deadlock detection and retry mechanisms. The ALAB must be accessed on each load and external coherence request. Furthermore, this structure requires significant changes to the out-of-order core's memory interface and pipeline control logic.

The LVQ is a FIFO structure, originally proposed as a simpler alternative to the ALAB, which records load values in program order from a leading execution and replays them for the trailing execution [88]. Architecting an LVQ within a CMP involves significant local and global changes to the processor core and CMP design, including:

1. **Modifications to the existing, heavily optimized processor/cache interface.** The trailing thread must bypass the cache and store buffer interface in favor of the LVQ, which adds bypass paths on the load critical path and additional functional unit resources schedule within the pipeline.

2. **Modifications to out-of-order scheduler.** The trailing thread only reads values in program order, which is a major policy change in front-end and out-of-order scheduling logic. The alternative, an out-of-order issue LVQ, eliminates the scheduling restriction, but has a similar complexity and area overhead as a multiported store buffer [72].

3. **High-bandwidth cross-core datapath.** Across processor cores, the LVQ requires a high-bandwidth path to transfer all load values and addresses. In order to avoid becoming a new performance bottleneck, this path must at least match the aggregate L1 cache read bandwidth. Furthermore, for the in-order version of the LVQ, the outcomes of all resolved branches must also be recorded and transferred for efficient, non-speculative re-execution. The LVQ datapath cannot use existing on-chip memory interconnects, because they are tuned to support L1 miss and on-chip coherence traffic, which is significantly less than the aggregate L1 hit traffic.

The LVQ also limits error coverage of memory operations: there is no way to verify that load bypassing and forwarding completed correctly because the trailing execution relies upon leading thread load values.

Relaxed Input Replication. Alternatively, in relaxed input replication, redundant threads independently send load requests to caches and store buffers, as in a non-redundant design. This avoids the added complexity of strict input replication and provides detection for soft errors in load forwarding and bypass logic. However, this means that redundant executions are susceptible to input incoherence.

There are two general methods for tolerating input incoherence in relaxed input replication: robust forward recovery and rollback recovery. Prior work entrusts a robust checker—a checker that is immune to faults—to resolve input incoherence. For example, DIVA checkers with dedicated caches [10] and slipstreamed re-execution [120] both allow the leading threads' load values to differ from the trailing threads'. However, these proposals do not address the possibility of data races in shared-memory multiprocessors and require complex additions (checker or slipstreamed cores) to support redundant execution. Alternatively, the naive rollback solution—simply retrying upon error detection—uses existing hardware support, but offers no forward progress guarantee. Because incoherent cache state or races can persist in the memory system, the same incoherent situation can occur again during re-execution. The Reunion execution model addresses this problem.

5.2.4 Output Comparison

The sphere of replication's boundary determines where outputs are compared. Two main choices have been studied in CMPs: (1) comparing outputs before the architectural register file (ARF), and (2) comparing before the L1 cache [40, 72]. In both cases, stores and uncached load addresses require comparison. For detection before the ARF, each instruction result must also be compared. The chosen design affects performance and the needed comparison bandwidth.

Output comparison affects performance at retirement. Serializing instructions, such as traps, memory barriers, and non-idempotent instructions, stall further execution until the serializing instruction has been compared. Both executions must complete and compare the serializing instruction before continuing.

Comparison bandwidth is another design factor in superscalar processors because multiple instructions may need comparison each cycle. Prior work proposes techniques to reduce bandwidth requirements. Gomaa et al. compare only instructions that end dependence chains in a lossless detection scheme [40]. They report bandwidth savings of roughly twenty percent over directly comparing each instruction result.

This thesis studies output comparison using architectural fingerprints. This study is limited to systems with comparison before the ARF because existing precise exception support [103] can then be used to recover before outputs become visible to other processors. Architectural fingerprints address the comparison bandwidth factor by compacting retiring values into a small hash value (e.g., 16 bits) with a negligible loss in coverage.

5.2.5 Fingerprints over On-Chip Interconnects.

This chapter evaluates fingerprints in Reunion using both fixed-latency, pipelined dedicated channels between pairs of cores and the on-chip memory interconnect to transfer fingerprints for comparison. Dedicated channels provide a predictable fingerprint comparison latency and avoid affecting the performance and operation of other system components, however they do have drawbacks:

- Dedicated channels have an area cost in terms of global cross-core ports and buses, which are unused in a non-redundant CMP.

- Dedicated channels limit the assignment of core pairs. Building dedicated channels between all pairs of cores increases quadratically with additional cores and is, therefore, unscalable and expensive in future CMPs. Instead, cores must be paired statically at design time.

As observed in LaFrieda et al. [56], the static core assignment has broad system-level implications. "Dynamic core coupling" in redundant execution on CMPs can alleviate many of these issues. System-level limitations of the static assignment include:

- Inefficient pairing of cores based on cross-die variability where higher performance can be obtained by matching F_{max} within a pair.

- Reduced tolerance to manufacturing defects and device wearout—one defective core also disables chosen cores its statically chosen partner, artificially reducing the effective lifetime of the overall chip.

- Limitations on run-time thermal and power management—a fixed pair can exceed the desired thermal or power envelope, particularly if they are in close proximity to each other.

Therefore, there is strong motivation for investigating alternative, more flexible and inexpensive architectural fingerprint comparison channels, which allows core pairs to be assigned at runtime.

The existing on-chip memory/cache interconnect on recent chip multiprocessors is already optimized for short, high-bandwidth, low-latency messages between on-chip components (for example, the Sun Niagara 2 CMP crossbar is expected to provide 270GB/s of core-cache bandwidth at 1.4GHz [76] and existing crossbars can sustain one message/destination port/core cycle [53]).

As part of a graduate computer architecture class project (under the direction of Smolens), Chellappa and de Mesmay study the feasibility of transferring fingerprints between cores over the existing CMP on-chip memory/cache interconnect [27]. In this design, fingerprints are transferred over existing ports and wide datapaths, usually reserved for cache block requests. The messages containing each fingerprint request compete for bandwidth with existing cache request and reply traffic. The bandwidth required depends on the frequency of fingerprint comparison—a fixed instruction interval in Reunion. There are two system-level performance factors that are affected by moving fingerprints to the on-chip interconnect. First, the existing cache miss traffic must compete with additional requests, potentially increasing the effective on-chip cache latency and decreasing

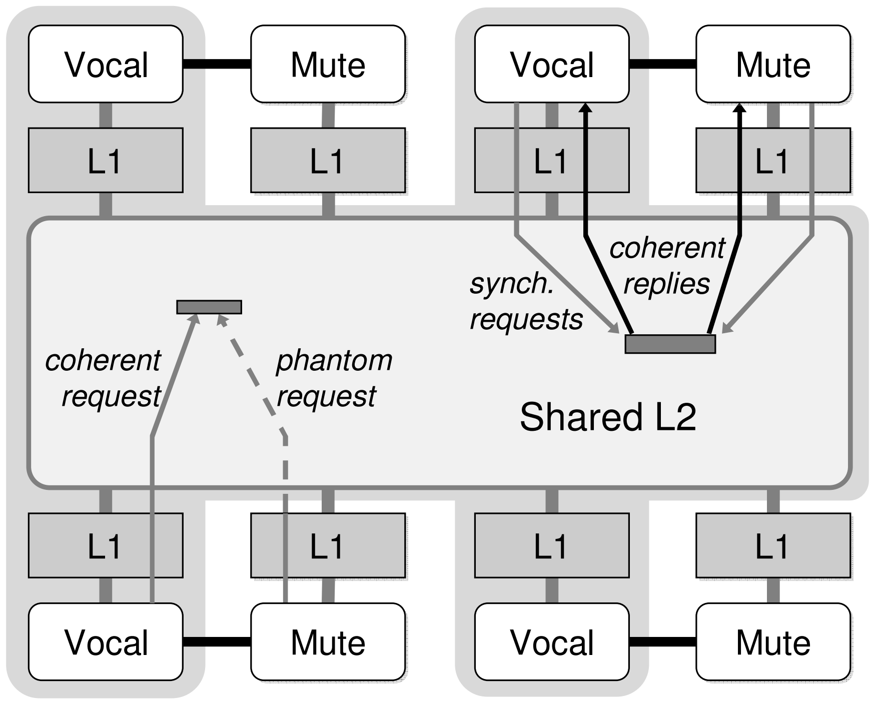


Figure 22. The Reunion architecture.

performance due to queuing at the cache controllers. Second, the fingerprints themselves must travel on a variable-latency interconnect which can extend resource occupancy (e.g., reorder buffer) for instructions waiting to be compared. This thesis shows that for reasonable fingerprint comparison intervals, the effects of each these two factors can be made negligible. Therefore, memory interconnects are suitable, while dedicated channels are over-engineered for fingerprint comparison.

5.3 Reunion Execution Model

This section presents a formal set of requirements for the Reunion execution model. The requirements provide redundant execution and relaxed input replication and allow reasoning about correctness independent of implementation. Figure 22 illustrates the concepts in this section.

5.3.1 System Definition

Definition 5.1. (Logical processor pair). A logical processor pair consists of two processor cores that execute the same instruction stream. To provide a single output from the sphere of replication, the logical processor pair presents itself as a single entity to the system.

The Reunion Execution Model differentiates the two cores as follows:

Definition 5.2. (Vocal and mute cores). Each logical processor pair consists of one vocal and one mute core. The vocal core exposes updated values to the system and strictly abides by the coherence and memory consistency requirements specified by the baseline system. The mute core never exposes updates to the system.

Vocal and mute cores use their existing private cache hierarchies and on-chip coherence protocol as in a non-redundant design. Definition 5.2 permits the mute core to write values into its private cache hierarchy, provided these values are not communicated to other caches or main memory.

Reunion uses redundant execution to detect and recover from soft errors that occur in program execution. We formally define safe execution as follows:

Definition 5.3. (Safe execution). Program execution is safe if and only if (1) all updates to architecturally-defined state are free of soft error effects, (2) all memory accesses are coherent with the global memory image, and (3) the execution abides by the baseline memory consistency model. Execution that is not safe is deemed unsafe.

The state that results from safe execution is:

Definition 5.4. (Safe state). The architectural state defined by the vocal core at a specific point in time is considered safe state if and only if it is free of soft errors; otherwise, the architectural state is deemed unsafe state.

5.3.2 Execution Model

Definition 5.2 requires that only the vocal abide by coherence and consistency requirements. Ideally, the mute core always loads coherent data values. However, precisely tracking coherent state for both vocal and mute is prohibitively complex (e.g., the coherence protocol must track two owners for exclusive/modified blocks).

Instead, Reunion maintains coherence for cache blocks in vocal caches, while allowing incoherence in mute caches. The mechanism for reading cache blocks into the mute cache hierarchy is the phantom request, a non-coherent memory request:

Definition 5.5. (Phantom request). A phantom request returns a value for the requested block without changing coherence state in the memory system.

The phantom request does not guarantee that the mute core will be coherent with the vocal, potentially leading to input incoherence within a logical processor pair:

Definition 5.6. (Input incoherence). Input incoherence results when the same dynamic load on vocal and mute cores returns different values.

Reunion requires vocal and mute to compare execution results as follows:

Definition 5.7. (Output comparison). Vocal and mute cores must compare all prior execution results before a value becomes visible to other logical processor pairs.

Lemma 5.1. *In the absence of soft errors, input incoherence cannot result in unsafe execution.*

Proof. If no soft error occurred during program execution, condition (1) of safe execution (Definition 5.3) is satisfied. If input incoherence occurred, the register updates and memory writes on the vocal still satisfy conditions (2) and (3). Therefore, safe execution results. □

Only undetected soft errors can result in unsafe state. Both input incoherence and soft errors can lead to divergent execution that must be detected and corrected. However, Lemma 5.1 proves that input incoherence alone cannot result in unsafe state.

5.3.3 Recovery

Definition 5.8. (Rollback recovery). When output comparison matches, the vocal's architectural state defines a new safe state that reflects updates from the compared instruction results; otherwise, rollback recovery restores architectural state to prior safe state.

Because only the vocal core's architectural state defines new safe state, Reunion requires a mechanism to initialize the mute core's architectural registers to match the vocal core.

Definition 5.9. (Mute register initialization). The vocal and mute cores provide a mechanism to initialize the mute core's architectural register file with values identical to the vocal's.

In the presence of input incoherence, naive retry cannot guarantee forward progress because the condition causing input incoherence can persist. Incoherent cache blocks in the mute's hierarchy can cause input incoherence until replaced by coherent values. Reunion addresses this problem with the synchronizing request:

Definition 5.10. (Synchronizing request). The synchronizing request returns the same, coherent value to both cores in the logical processor pair.

This definition differs from the original Reunion paper [104], which states that the synchronizing request must return "*a single* coherent value to both cores". This more relaxed definition allows for simpler implementations of the synchronizing request. For example, an implementation can attempt to return the same value to both cores using incoherent phantom requests and replies, as opposed to a requiring the same value to always be returned to both cores.

Mute register initialization and the synchronizing request are combined to construct the re-execution protocol and then prove that the protocol guarantees forward progress following rollback recovery.

Definition 5.11. (Re-execution protocol). After rollback recovery, the mute architectural register file is initialized to the values from the vocal. The logical processor pair then executes subsequent instructions non-speculatively (single-step), up to and including the first load or atomic memory operation. This operation is issued by both cores using the synchronizing request. After successful output comparison following this instruction, the logical pair resumes normal execution.

Lemma 5.2. (Forward Progress). *The Reunion re-execution protocol always results in forward progress.*

Proof. Rollback recovery is triggered either by a soft error, which does not persist, or by input incoherence, which can persist. In the first case, re-execution eliminates the error and results in successful output comparison. In the second case, the mute register initialization and synchronizing request guarantee safe execution and safe state to the first load. □

This proof depends upon the synchronizing request providing a guarantee of forward progress. Simpler implementations of the synchronizing request are also considered in Section 5.4. These

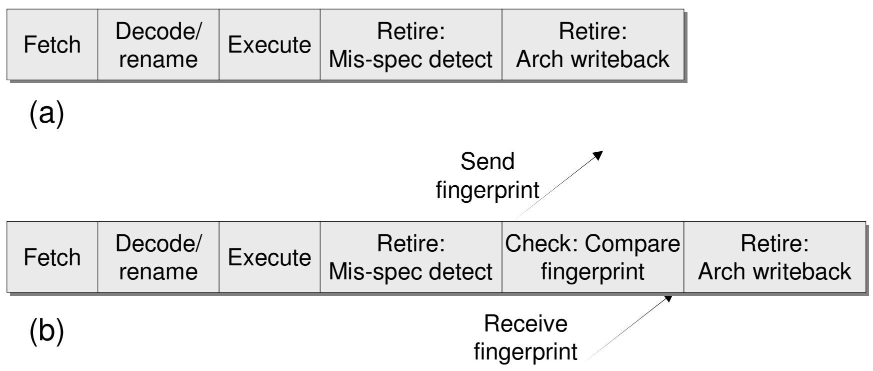


Figure 23. (a) Baseline pipeline and (b) a pipeline fingerprint checks before retirement.

simpler implementations have a high probability of succeeding, but cannot provide a complete guarantee.

An implementation must provide the required behaviors of the execution model, but the system designer has latitude to optimize. In Section 5.4, a fast re-execution protocol implementation handles common case re-execution, while a slower version implements the rarely needed register file copy. The high-level tradeoffs for checkpoint and recovery implementation are also discussed.

5.4 Reunion Microarchitecture

This section first describes the baseline CMP and processor microarchitecture. The section continues with the changes required to implement the Reunion execution model in a shared cache controller and processor core.

5.4.1 Baseline CMP

Cache Hierarchy. This chapter assumes a baseline CMP with caches similar to Piranha [15]. A shared cache backs multiple write-back L1 caches private to each processor core. The shared cache controller accepts memory requests from all cores, coordinates on-chip coherence for blocks in private caches, and initiates off-chip transactions. The Reunion execution model can also be implemented at a snoopy cache interface for microarchitectures with private caches, such as Montecito [64].

Processor Microarchitecture. This chapter assumes the simplified out-of-order processor pipeline illustrated in Figure 23(a). Instructions are fetched and decoded in-order, then issued, executed, and written back out-of-order. In-order retirement stages inspect instructions for branch misspeculation and exceptions, and write instruction results to the architectural register file, as in the Pentium-M [94] described in Section 3.1. Stores initially occupy a speculative region of the store buffer. At retirement, the stores transition to a non-speculative region of the store buffer and drain to the L1 cache.

This chapter assumes single-threaded processor cores. Reunion can benefit from the efficient use of otherwise idle resources in simultaneous multithreaded designs; however, cores must run only vocal or mute threads to prevent vocal contexts from consuming incoherent cache blocks.

5.4.2 Shared Cache Controller

The shared cache controller is responsible for implementing the vocal and mute semantics, phantom requests, and synchronizing requests. As in non-redundant designs, the shared cache controller maintains coherence state (e.g., ownership and sharers lists) for all vocal cores.

Because coherence is not necessary in mute caches, sharers lists never include mute caches and mute caches can never become exclusive or modified block owners. The coherence protocol behaves as if mute cores were absent from the system. To prevent values generated by mutes from being exposed to the system, the shared cache controller ignores all eviction and writeback requests originating from mute cores.

Phantom requests. All non-synchronizing requests from the mute to the shared cache controller are transformed into phantom requests. The phantom request produces a reply, although the value need not be coherent, or even valid. Phantom replies grant write permission within the mute hierarchy.

The phantom request allows several strengths, depending on how diligently it searches for coherent data. The weakest phantom request strength, a *null* phantom request, returns arbitrary data on any request (i.e., any L1 miss). While trivial to implement, null has severe performance implications. A *shared* phantom request returns an existing cache block value for hits in the shared cache, but returns an arbitrary value on misses. Finally, the *global* phantom request achieves the

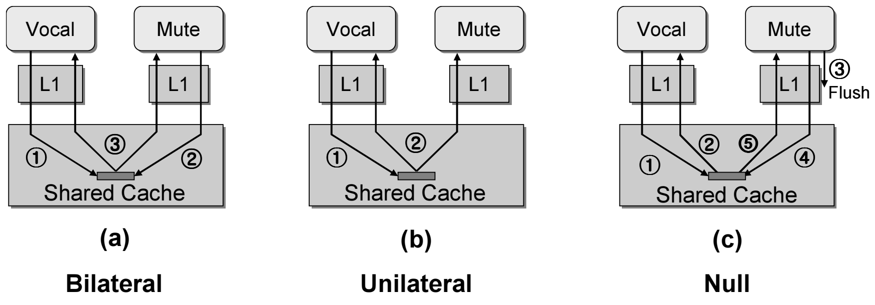


Figure 24. Three forms of synchronizing request.

best approximation of coherence. This request not only checks the shared cache, but also private vocal caches and issues read requests to main memory for off-chip misses. In terms of complexity, this is a small departure from existing read requests. Unless otherwise noted, this chapter assumes global phantom requests.

Synchronizing Requests

This section now presents several possible instantiations of the synchronizing request and their tradeoffs. The key task of a synchronizing request is to (1) return the same, coherent value to both the vocal and mute cores and (2) flush the incoherent cache block from the mute cache. The synchronizing requests described in this section dominate the recovery latency and are generally comparable to a shared cache hit in terms of latency.

The Reunion paper originally specified the bilateral synchronizing request, however industry feedback indicated that simpler mechanisms are greatly preferable and very rare failures can be acceptable [83].

Bilateral Synchronizing request. The bilateral synchronizing request uses the shared cache controller to enforce coherence between the vocal and mute cores, as illustrated in Figure 24(a). The bilateral request first flushes the block from private caches (returning the vocal's copy to the shared cache, while discarding the mute's). When both requests have been received at L2, the shared cache controller initiates a coherent write transaction for the cache block on behalf of the pair. This obtains sufficient permission to complete instructions with both load and store semantics. After obtaining the coherent value, the shared cache controller atomically replies to both the vocal and mute cores.

This form of the synchronizing request requires unique functionality in the shared cache controller to (1) wait for an unsolicited request message from another core and (2) atomically send two replies to a pair of cores. The former requires careful design to avoid races with other concurrent requests for the same cache block, which increases the complexity of the shared cache controller. This feature also increases the validation effort of the shared cache controller. The latter feature is a small change from existing shared cache controller functionality, because messages sent to multiple destinations are already necessary for sending invalidation requests on shared blocks.

Unilateral Synchronizing Request. Unlike the bilateral request, the unilateral synchronizing request comprises a single request and an atomic reply pair, initiated solely by the vocal core (shown in Figure 24(b)). The request causes the block to be flushed from the vocal's caches, while the reply flushes the value from the mutes'. The mute core must store the reply until it has reached the point at which it can consume the value.

The intent of this request is to simplify the implementation of the synchronizing request at the shared cache. Instead, the unilateral request moves complexity towards the processor core, although the change is arguably limited in scope. The addition is that the mute core must accept and buffer an unsolicited reply message. Because only one synchronizing request can be outstanding per logical processor pair, this buffer need only be one entry. However, because mute's execution progress can race with this reply message (e.g., the mute re-execution can arrive at the first load before or after the reply has been received), this value must be carefully delivered to the appropriate instruction.

Null Synchronizing Request. An alternative approach—which fits cleanly into the spirit of the Reunion execution model—is the null synchronizing request. This request uses the existing coherent load from the vocal core. The mute core, in contrast, issues a sequence of a flush request to remove the incoherent block from its private caches, followed by its typical phantom request to return the block to the mute's cache.

The key advantage of this synchronizing request is its simplicity. It uses mechanisms already needed to implement Reunion. However, the null synchronizing request cannot guarantee that it will always return the same value to both cores. Therefore, this synchronizing request may have to be invoked multiple times in the case of persistent data races on heavily contended blocks. This has been shown possible with tightly coded micro-benchmarks and short shared cache latencies (e.g., deadlock only occurs if the L2 access latency is dialed down to one cycle), but in practice works well on both commercial and scientific workloads.

The evaluation in this thesis shows that not only are data races rare, but also that repeated races are rare, therefore a slow mechanism for recovery is acceptable. A retry mechanism can include exponential back-off to avoid repeated races. Alternatively, a "hammer" technique can guarantee forward progress. For example, the shared cache controller can temporarily block new requests from other cores in order to avoid further races. Thus, the common case synchronizing request is fast, while the uncommon case of repeated races can be resolved correctly.

5.4.3 Processor Pipeline

This section now describes the processor pipeline changes for the Reunion execution model, output comparison, and recovery.

Safe state. The vocal processor core maintains safe state in the ARF, non-speculative store buffer, and memory. Safe state can always be reached by the vocal by (1) retiring all instructions that have completed output comparison without error to the architectural register file and the non-speculative store buffer and (2) flushing all uncompared instructions from the pipeline (e.g., precise exception rollback).

Output comparison. Instruction outputs must be compared before retiring to architectural state. The key addition is an in-order retirement stage called check, as shown in Figure 23(b). Check first generates a fingerprint a hash of instruction results from the entering instructions [105]. Check then compares its fingerprint with the partner core's fingerprint to detect differences. A matching fingerprint comparison retires the instruction and writes the instruction results to safe state in the architectural register file. A mismatch invokes recovery. Instructions cannot enter check speculatively; they must be guaranteed to retire if the instruction results match.

Logically, the fingerprint captures all register updates, branch targets, store addresses, and store values. The number of instructions summarized by each fingerprint is a design parameter called the

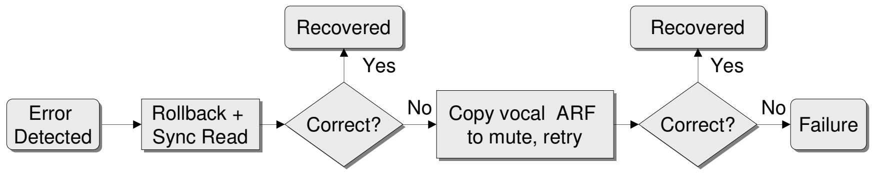


Figure 25. The re-execution protocol.

fingerprint interval; longer comparison intervals need proportionally less comparison bandwidth. At the end of each fingerprint interval, each core sends its fingerprint to the partner core for comparison.

An analytic performance model of the microarchitectural effects of the fingerprint interval and fingerprint comparison latency is presented later in this section. The model and empirical evidence show that the performance difference between intervals of one and fifty instructions is insignificant in our workloads, despite increased resource occupancy, because useful computation can continue to the end of the interval.

The time required to generate, transfer, and compare the architectural fingerprint is combined into a parameter called the comparison latency. Because the vocal and mute cores swap fingerprints, the comparison latency is the one-way latency between cores. This latency overlaps with useful computation, at the cost of additional resource occupancy. The observed comparison latency, however, can be extended because the two cores are only loosely coupled in time. While the vocal and mute execute the same program, their relative progress can drift slightly in time, due to contention accessing shared resources (e.g., the L2 cache) and different private cache contents.

Re-execution. Upon detection of differences between the vocal and mute, the logical processor pair starts the re-execution protocol illustrated in Figure 25. To optimize for the common case, the protocol is divided into two phases. The first handles detected soft errors and detected input incoherence errors. The second phase addresses the extremely rare case where results of undetected input incoherence retire to architectural registers.

Both vocal and mute cores invoke rollback-recovery using precise exception support and, in the common case, restore to identical safe states in their architectural register files. Both cores then non-speculatively single-step execution up to the first memory read. Each core then issues a synchronizing memory request eliminating input incoherence for the requested cache block and compares a fingerprint for all instructions in the interval. Following comparison, the re-execution protocol has made forward progress by at least one instruction. The cores then continue normal speculative, out-of-order execution.

If the first phase fails output comparison, the second phase starts. The vocal copies its architectural register file to the mute and the pair proceeds with re-execution, as in the first phase. Because the vocal core always maintains safe state in the absence of soft errors, this will correctly recover from all incoherence errors. If the cause was a soft error missed by fingerprint aliasing, the protocol cannot recover safe state and therefore must trigger a failure (e.g., detected, uncorrectable error interrupt). The re-execution protocol can be implemented in microcode.

External interrupts. External interrupts must be scheduled and handled at the same point in program execution on both cores. Fingerprint comparison provides a mechanism for synchronizing the two cores on a single instruction. Reunion handles external interrupts by replicating the request to both the vocal and mute cores. The vocal core chooses a fingerprint interval at which to service the interrupt. Both processors service the interrupt after comparing and retiring the preceding instructions.

Hardware cost. Fingerprint comparison requires queues to store outstanding fingerprints, a channel to send fingerprints to the partner core, hash circuitry, and a comparator. The fingerprint queues can be sized to balance latency, area, and power. The check stage delays results being written into the architectural register file. The results can be stored in a circular buffer during the check stage or read again at retirement.

5.4.4 Serializing Check Overhead

Instructions with serializing semantics such as traps, memory barriers, atomic memory operations, and non-idempotent memory accesses impose a performance penalty in all redundant execution microarchitectures that check results before retirement. Serializing instructions must stall pipeline retirement for an entire comparison latency, because (1) all older instructions must be compared and retired before the serializing instruction can execute and (2) no younger instructions can execute until the serializing instruction retires.

Upon encountering a serializing instruction, the fingerprint interval immediately ends to allow older instructions to retire. The fingerprint is updated to include state that must be checked before executing the serializing instruction (e.g., uncacheable load addresses). Once the older instructions retire, the serializing instruction completes its execution and check. This comparison exposes timing differences between the cores (due to loosely-coupled execution) and the entire comparison latency. Normal execution continues once the serializing instruction retires.

5.4.5 Fingerprint comparison interval and latency: analytic model

This section presents an analytical model that explains the performance impact of adding architectural fingerprint comparison and the instruction check stage to a modern wide-issue pipeline. The independent fingerprint design parameters in the model are the fingerprint comparison latency (L_{fp}) in clock cycles and the fingerprint comparison interval (I_{fp}) in instructions. This analysis assumes dedicated, pipelined fingerprint comparison channels with a fixed transmission and comparison latency. Furthermore, the model assumes the fetch stage is aggressive enough to saturate the re-order buffer and that serializing instructions are absent.

The model depends on several pipeline parameters:

- $ROBSize$: the number of instructions in the re-order buffer
- L_{pipe}: the length of the pipeline for which instructions occupy ROB entries (including retirement and check stages). This parameter is fixed at $L_{pipe} = 5$ stages in this study.[1]
- R: the average instruction retirement rate in a pipeline without fingerprint comparison (a property of the workload and the pipeline width).

This thesis first derives two performance models to account for the fingerprint comparison latency and interval separately. Next, this thesis presents a single model accounting for both parameters.

A performance model for a pipeline with fingerprint comparison on every instruction ($I_{fp} = 1$) is derived. Because fingerprint comparison can only slow down single-threaded execution, performance is bounded by the minimum of performance derived from Little's Law (relating population

[1]Five stages matches the CMP simulator's minimum pipeline depth, excluding the in-order fetch/decode/rename stages and is representative of the out-of-order sections recent industry pipelines.

in instructions and occupancy time to throughput) and the original program's IPC without fingerprint comparison. In this model, performance is only limited when the ROB fills with instructions waiting for retirement or the fingerprint comparison.

$$IPC_{min\_interval} = \min\left(\frac{ROBSize}{L_{pipe} + L_{fp}}, R\right) \tag{5.1}$$

Next, a performance model for a pipeline that compares intervals of instructions, but has an instantaneous latency for fingerprint comparison ($L_{fp} = 0$) is derived. If the fingerprint comparison interval (I_{fp}) is greater than one instruction, older instructions entered the check stage but cannot be compared until enough subsequent younger instructions also enter check and complete the interval. For a program nominally executing R instructions per cycle, this additional wait adds $\frac{I_{fp}}{R}$ cycles to execution.

$$IPC_{min\_latency} = \min\left(\frac{ROBSize}{L_{pipe} + \frac{I_{fp}}{R}}, R\right) \tag{5.2}$$

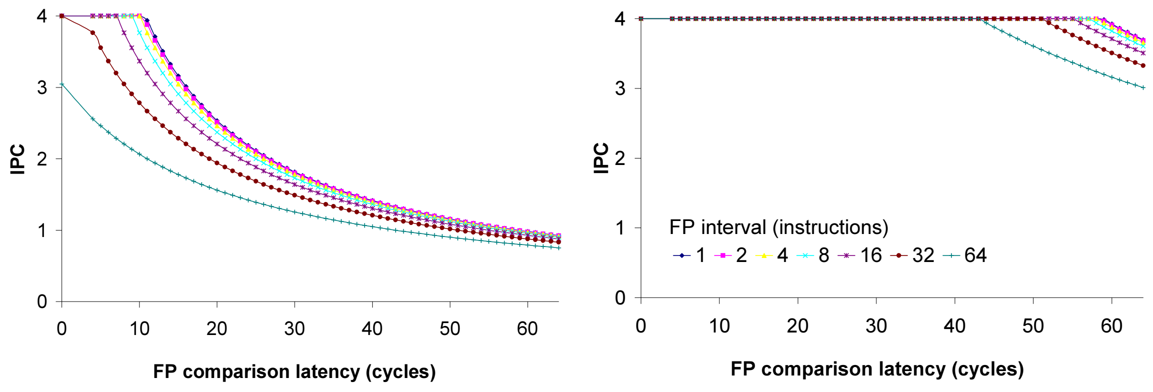
Note that this model is accurate only for I_{fp}'s that are divisors of $ROBSize$. There is a significantly more complicated relationship for fingerprint intervals that do not cleanly fit in the ROB. Unfortunately, while this is trivial to simulate, a closed-form algebraic relationship that can lend insight towards the pipeline's performance is not known.

Finally, the added instruction occupancy contributions of both factors are combined in the comprehensive performance model shown below.

$$IPC = \min\left(\frac{ROBSize}{L_{pipe} + L_{fp} + \frac{I_{fp}}{R}}, R\right) \tag{5.3}$$

The results of this model are illustrated in Figure 26 for four-wide pipelines with ROB sizes of 64 and 256 entries, running a workload that can saturate the pipeline. This model is also limited to I_{fp} that evenly divide the $ROBSize$. The results show that as the fingerprint comparison latency increases, it dominates performance, while the fingerprint comparison interval's effect is dampened because it adds latency more slowly.

Furthermore, the right-hand figure shows that the 256-instruction ROB provides sufficient buffering to hide virtually all performance effects of fingerprint comparison for reasonable on-chip trans-



(a) (b)

Figure 26: Analytic performance model of the effects of fingerprint comparison for a 4-wide pipeline with $L_{pipe} = 5$ cycles, $R = 4$ IPC and ROB sizes of (a) 64 and (b) 256 entries.

mission and comparison latencies (10's of cycles) and a range of fingerprint intervals. By contrast, the left-hand figure shows significant sensitivity to both fingerprint comparison latency and interval. With long latencies or intervals, the entire ROB ends up buffering completed instructions that are waiting to be checked. However, in practice, the effect of these results on the 64-entry ROB are tempered because, despite the availability of four-wide pipelines, many real-world workloads only achieve IPCs of 0.2-2 [3, 14, 31] and typically only occupy small fractions of the ROB. These workloads already typically (1) can overlap the additional checking latencies caused by fingerprint comparison and (2) leave adequate buffering for instruction checking. Therefore, the actual impact of architectural fingerprints—even with smaller ROBs—is typically small. This result is empirically shown in Section 5.5.

Finally, these results have been validated against a simulation model, showing performance results within 1% over practical ranges for these parameters (latencies from 0 - 1,024 cycles, intervals from 1 to $ROBSize$).

5.4.6 Lock Primitive Implementation

This section describes the implementation of lock primitives in a Reunion design. The implementation of atomic memory operations such as Sun's read-modify-write (RMW), compare-and-swap, test-and-set, and fetch-and-increment is first described, followed by the special considerations required for two-phase locking primitives, such as MIPS's load-locked and store-conditional

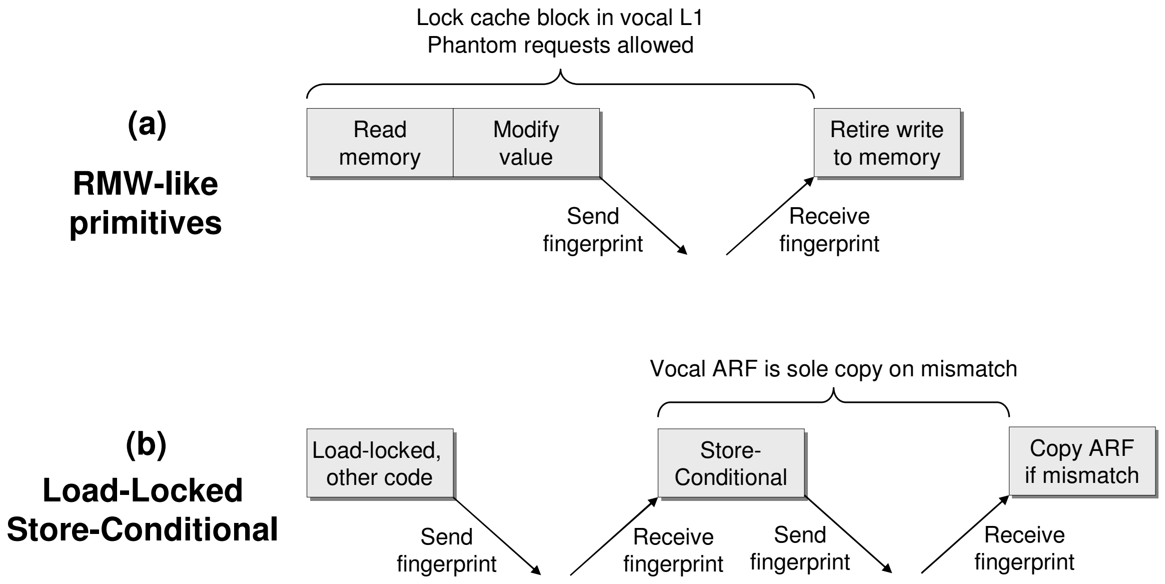


Figure 27. Lock implementation.

(LL/SC) instruction pair.

Atomic memory operations. Atomic memory operations consist of three basic tasks that occur atomically, as observed by other processors through shared memory: (1) one or more loads from memory, (2) an equality test or arithmetic operation on the loaded value, and (3) a store to memory and memory barrier. In order to succeed, the tasks must occur without any intervening store by other processors in the system. Unlike non-redundant multiprocessors, where the determination of whether the locks primitives succeed (or fail by succumbing to a data race) can be made locally at a single processor core, this process is distributed across the vocal and mute cores in Reunion. Hence, the implementation must ensure that both the vocal and mute cores agree on the outcome of the operation before it is exposed to other logical processor pairs by the vocal core. This requirement is relaxed in the discussion of the LL/SC implementation.

The key problem for atomic operations introduced by the Reunion execution mode is that the fingerprint comparison adds latency between tasks (2) and (3), as illustrated in Figure 27(a). The instruction must complete a fingerprint comparison before it can complete task (3), because that action architecturally exposes results to other processors. The Reunion execution model introduces a new requirement in the lock's execution: both tasks and the fingerprint comparison must be performed without permitting intervening stores from other processors, as this extends the window of vulnerability during fingerprint comparison.

The solution to the window of vulnerability during atomic operations proposed and evaluated in this work is to lock the vocal L1 cache during the atomic operation. L1 cache locking involves delaying the processing of external requests for the cache block during the fingerprint comparison. Cache locking—globally, on a set, or on a single block basis—is a well-known technique for temporarily preventing other processors from accessing a locally cached block.[2] Locking within a single on-chip cache has been part of the Pentium Pro [62] and support continues in today's microarchitectures [45].

Locking the cache in a redundant system is not without dangers, however. Because both the vocal and mute cores must read the same value—and the vocal's private L1 cache can supply that value to the mute—it is possible to create a deadlock scenario if the vocal's L1 cache is locked too early. This situation arises if the mute L1 does not contain the block (and therefore, must issue a phantom request) and the vocal L1 contains exclusive ownership of the block and has locked out external requests. The cycle is as follows: the vocal core will not unlock until the fingerprint comparison completes, while the partner mute core cannot send a fingerprint until its phantom request to the locked cache has completed.

The deadlock can be avoided in at least two ways. First, an aggressive implementation of the L1 cache controller can continue processing phantom request messages while the cache is locked. Because phantom requests do not change coherence state and can return non-coherent values, they can be safely processed at any time. This prevents the circular dependence between the mute's phantom request and fingerprint comparison. Alternatively, the deadlock can be avoided entirely by immediately replying to phantom requests targeting a locked cache with an arbitrary (poisoned) value. This incurs a fingerprint mismatch, but does not require the cache to process phantom requests when locked. In the remainder of this work, we evaluate Reunion using the first implementation.

Load-locked and store-conditional. Several prominent instruction sets choose to provide an LL/SC instruction pair as the architected synchronization primitive [68, 30]. Unlike the Pentium Pro, processors implementing this primitive shun locking the cache to avoid memory requests from other processors, preferring instead to set a (unfortunately named in this context) "lock bit" upon

[2]If fingerprinting across the interconnect is used, we assume that fingerprint messages continue to be processed out-of-band, while the cache is locked.

Table 9. Outcomes for different races during the store-conditional phase of an LL/SC pair.

Vocal	Mute	Outcome
Fails	Fails	Fingerprint matches, no race
Fails	Succeeds	Fingerprint mismatch; fails architecturally, copy vocal ARF
Succeeds	Fails	Fingerprint mismatch; succeeds architecturally, copy vocal ARF
Succeeds	Succeeds	Fingerprint matches, no race

executing a load-locked instruction. The lock bit is reset by any intervening external access to the load-locked's effective address before a corresponding store-conditional instruction (the store is architecturally converted to a NOP if the bit has been reset before the store commits). Because these architectures avoid locking the cache for synchronization primitives, the technique described for atomic memory operations cannot be applied.

Instead, an alternative approach can apply the principles of the Reunion execution model to the lock's execution, but trades soft error vulnerability and performance to avoid the hardware cost of cache locking. This approach takes advantage of an outcome of the vocal core: that in the absence of soft errors, the vocal core always performs cache-coherent and memory model-consistent actions (Lemma 5.1). This allows the vocal to determine solely the outcome of the LL/SC operation.

The approach is illustrated in Figure 27(b). Before executing the store-conditional, the results of the load-locked instruction and following instructions are compared via fingerprint comparison. This ensures that both vocal and mute cores loaded the same value and checks the execution of the body of the lock code. Second, the store conditional is executed and committed on both cores and the result of that operation—whether the store conditionally succeeded or failed—is compared via fingerprint between the two cores. The store conditional must be committed, because its condition is only known once the instruction commits.

The cores can agree or disagree on the outcome. The cases are enumerated in Table 9. If the cores agree, redundant execution can continue. However, if the cores disagree, Lemma 5.1 is applied to restore consistent architectural state to both cores by copying the vocal's architectural register file to the mute, as done in the second phase of the recovery protocol. In effect, this approach allows the vocal core to perform the store-conditional and inform the mute of the outcome. Furthermore, this approach provides the same forward progress guarantees as the baseline system when using LL/SC.

However, this approach opens a soft error vulnerability during the period in which the vocal executes the store-conditional. If a soft error were to affect the vocal's operation during this time, the execution can result in detectable, uncorrectable errors or silent data corruption. Because this period is only a small fraction of overall execution time, this is unlikely to have a significant impact on the overall system's soft error vulnerability.

This approach can also be adapted for use with atomic memory operations with the same tradeoffs on soft error vulnerability and performance. Because the impact of data races is shown to be rare in the evaluation, this thesis does not separately evaluate the performance effect of this approach.

Another option for implementing LL/SC is to depend solely upon the vocal core to determine the outcome of the SC, transmit this information to the mute core in a fingerprint, and have it mirror the execution of the SC request (this information is implicitly transferred in the approach discussed above). This process simplifies lock implementation and provides the same soft error vulnerability as the previously discussed approach. Because the vocal core's outcome will be used regardless input incoherence or a soft error—both schemes are vulnerable to undetected soft errors during the SC execution on the vocal—there is no reliability benefit to checking the mute's execution of the SC.

5.4.7 Checkpointing and Re-execution

This section explores the high-level performance tradeoffs for various checkpoint and recovery models. Checkpoint mechanisms are explored along two axes: the scope of the checkpoint—local per-processor and system-wide global checkpoints—and the checkpoint interval length—short ROB-sized checkpoints of tens to hundreds of instructions and long checkpoints containing thousands of instructions. Two workload classes are explored: phased workloads such as barrier-based scientific programs and decision support queries and phase-less workload such as OLTP [42] and web server commercial workloads. The tradeoffs are illustrated graphically in Figure 28. Experiences with Reunion suggest that the desirable third quadrant—local checkpoints with short comparison intervals—are both practical with today's microarchitectures and provide a low execution time overhead.

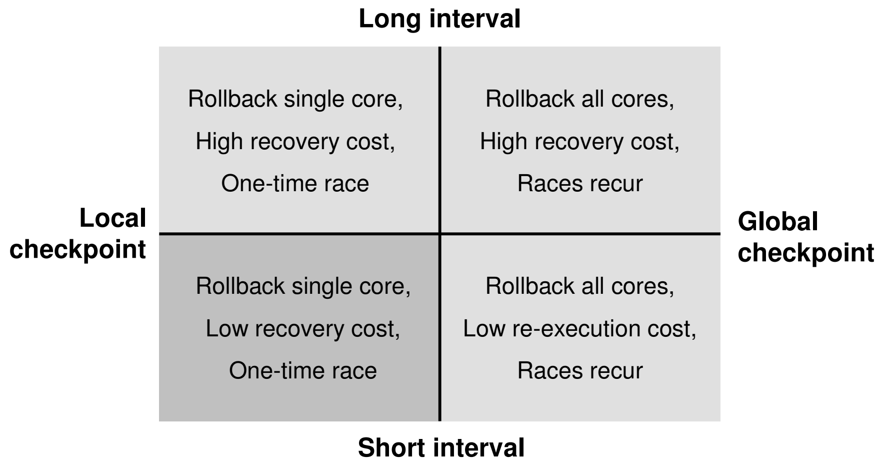


Figure 28: The tradeoffs in the checkpoint and recovery design space. Local checkpoints with short checkpoint intervals provide the lowest overall recovery overhead.

Checkpoint scope. Local checkpoints are attractive if recovery is frequent, there are a large number of processors, or if the recovery process is likely to repeat.

With global checkpoints, the state of all processor cores in the system is simultaneously recorded, while with local checkpoints, the architectural state of individual processor cores is separately stored. Global checkpoints linearly increase the cost of recovery. As the system size increases, global checkpoint recovery requires discarding and then re-executing proportionally more work. By contrast, local checkpoints require discarding and re-executing the work of only a single processor core.

Furthermore, global checkpoints restore the entire system to a state that was previously encountered. In the case of recovery due to races in phased-based workloads, this a particularly unfortunate situation. Phase-based workloads can exhibit well-defined points in execution where input incoherence-causing data races are likely to re-occur. When a data race occurs in a global checkpoint-based system, all processors are recovered to an identical point prior to the race. Unfortunately, this also recreates the precise conditions for the race to recur (this can also affect phase-less workloads which are otherwise unlikely to repeatedly encounter the same race again). With the Reunion re-execution protocol as stated above, this process produces one additional load's worth of forward progress, and then the race scenario repeats itself again! This situation has the compound cost of repeated recovery, each time discarding the work of all processors and can greatly affect performance [56].

By contrast, local checkpoints only discard the work of individual processor cores—greatly reducing the amount of work that needs to be recomputed during recovery. Furthermore, recovery does not recreate the same global race situation as before. Hence, the recovery with local checkpoints is cheaper and less likely to be repeated (particularly in phase-less workloads where the operations on other processors are uncorrelated with the recovering processor).

Checkpoint interval. As with the checkpoint scope, the interval between checkpoints also determines the amount of work that must be discarded and reexecuted following a recovery. Short checkpoint intervals incur a small tolerable penalty, while longer intervals can significantly increase the recovery overhead (particularly if races recur) [56].

The checkpoint interval is primarily determined by the costs of error detection and taking a checkpoint. For a traditional out-of-order superscalar processor such as Pentium-M, a per-instruction checkpoint granularity is feasible because of existing precise exception support, which already requires the processor to stop architectural execution at any instruction boundary.

Checkpoint intervals must be equal to or longer than the comparison intervals. This is because, for the purposes of error detection, no additional benefits are accrued from checkpointing more frequently than errors are detected because intermediate checkpoints will never be used for recovery. For architectural fingerprint-based error detection, the primary system-level limit is the bandwidth required for transferring fingerprints between cores [105]. With dedicated comparison channels, comparison can feasibly done every two to four instructions, while with fingerprints over an interconnect, the interval must be longer to minimize the performance impact on the on-chip memory system.

Alternatively, checkpoint interval can be driven by a more expensive checkpoint mechanism, such as copying the register file (as with Intel's C6 architectural checkpoints [35] or academic proposals [51, 56, 108]). In these designs, the checkpoint interval must be increased to amortize the time spent stopping the system while creating the checkpoint or performing error detection.

Table 10. Simulated baseline CMP parameters.

Processor cores	4 logical processors, UltraSPARC III ISA 4 GHz 12-stage pipeline; out-of-order 4-wide dispatch / retirement 256-entry RUU; 64-entry store buffer
L1 cache	64KB split I/D, 2-way, 2-cycle load-to-use, 2 read, 1 write ports, 64-byte lines, 32 MSHRs
Shared L2 Cache	16MB unified, 4 banks, 8-way, 35-cycle hit latency, 64-byte lines, crossbar to L1s, 64 MSHRs
ITLB	128 entry 2-way; 8K page
DTLB	512 entry 2-way; 8K page
Memory	3GB, 60ns access latency, 64 banks

5.5 Evaluation

This thesis evaluates Reunion using Flexus, which provides cycle-accurate, full-system simulation of a chip multiprocessor [118]. Flexus extends Virtutech Simics with cycle-accurate models of an out-of-order processor and memory system.

A CMP with four logical processors is simulated: four cores for non-redundant models and eight cores for redundant models. On-chip cache bandwidth is assumed to scale in proportion with the number of cores. The CMP model uses a cache hierarchy derived from Piranha [15]. For Reunion, the vocal and mute L1 cache tags and data are independently modeled. System parameters are listed in Table 10.

Table 11 lists the commercial and scientific application suite. All workloads run on Solaris 8. The commercial workloads include TPC-C v3.0 OLTP on two commercial database management systems, IBM DB2 v8 Enterprise Server Edition, and Oracle 10g Enterprise Database Server. The database is tuned to maximize performance of the non-redundant system model.

Three representative queries from the TPC-H decision support system (DSS) workload are selected, showing scan-dominated, join-dominated, and mixed behavior. Web server performance is evaluated with the SPECweb99 benchmark on Apache HTTP Server v2.0 and Zeus Web Server v4.3. The server performance of web servers saturated by separate clients over a high-bandwidth link is reported. The scientific workloads are four parallel shared-memory scientific applications that exhibit a range of memory access patterns.

Table 11. Simulated workload parameters.

Commercial Workloads	
DB2 OLTP	100 warehouses (10GB), 64 clients, 2GB BP
Oracle OLTP	100 warehouses (10GB), 16 clients, 1.4GB SGA
DB2 DSS	Qry 1 (scan); Qry 2 (join); Qry 17 (balanced)
	100 warehouses (10GB), 2GB BP
Apache Web	16K connections, fastCGI, worker thread model
Zeus Web	16K connections, fastCGI
Scientific Workloads	
em3d	768K nodes, degree 2, span 5, 15% remote
moldyn	19,652 molecules, boxsize 17, 2.56M max iters.
ocean	258x258 grid, 9600s relax, 20K res., errtol 1e-7
sparse	4096x4096 matrix

This evaluation employs a sampling approach that draws many brief measurements over 10 to 30 seconds of simulated time for OLTP and web applications, the complete query execution for DSS, and a single iteration for scientific applications. 95% confidence intervals of +/-5% error on change in performance are targeted using matched-pair comparison [118]. Measurements launch from checkpoints with warmed caches and branch predictors, then run for 100,000 cycles to warm pipeline and queue state prior to 50,000 cycles of measurement. Aggregate user instructions committed per cycle are reported as the performance metric, which is proportional to overall system throughput. Fingerprints are compared on every instruction. Soft faults are not injected, however input incoherence events, output comparison, and recovery are modeled in detail.

5.5.1 Baseline Performance

This section evaluates the baseline performance of redundant execution in a CMP for a representative system using strict input replication ("Strict") and Reunion.

Strict models a system with strict input replication, fingerprint comparison across cores for error detection, and recovery within the ROB (as described for Reunion in Section 5.4.3). Strict serves as an oracle performance model for all strict input replication designs with recovery. It imposes no performance penalty for input replication (e.g., lockstepped processor cores or an LVQ with no resource hazards). However, the model includes the penalties from buffering instructions during check. The Reunion model demonstrates the performance of relaxed input replication and fingerprint comparison and recovery. To support recovery within the speculative window, both

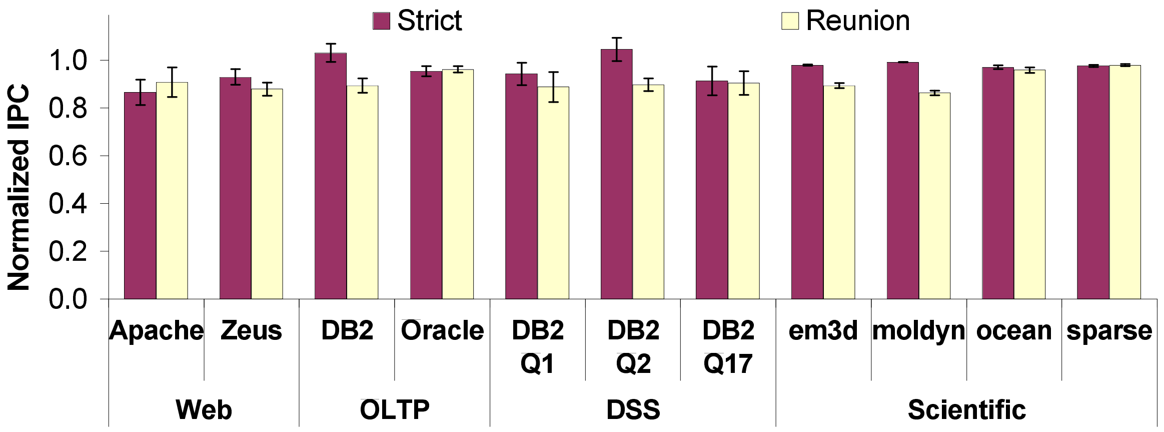


Figure 29: Baseline performance of redundant execution with strict input replication and Reunion, normalized to a non-redundant CMP, with a 10-cycle comparison latency.

systems check instruction results before irrevocably retiring them to the architectural register file and non-speculative store buffer.

Figure 29 shows the baseline performance of both models normalized to the performance of a non-redundant baseline CMP, with a ten-cycle comparison latency between cores. As compared to the non-redundant baseline, the strict model has a 5% and 2% average performance penalty for commercial and scientific workloads, respectively, while Reunion shows 10% and 8% average respective performance penalties. The low performance overhead of Reunion demonstrates that relaxed input replication is a viable redundant execution model. In the following sections, explore the performance of these execution models is explored in more detail.

5.5.2 Checking Overhead

This section first examines the performance of Strict to understand the performance penalties of checking redundant executions across cores in a CMP. First, serializing instructions cause the entire pipeline to stall for the check because no further instructions can execute until these instructions complete. The check fundamentally extends this stall penalty: as the comparison latency increases, the retirement stalls must also increase. Second, pipeline occupancy increases from instructions in check occupying additional ROB capacity in the speculative window. For workloads that benefit from large instruction windows, this decreases opportunities to exploit memory-level parallelism (MLP) or perform speculative execution.

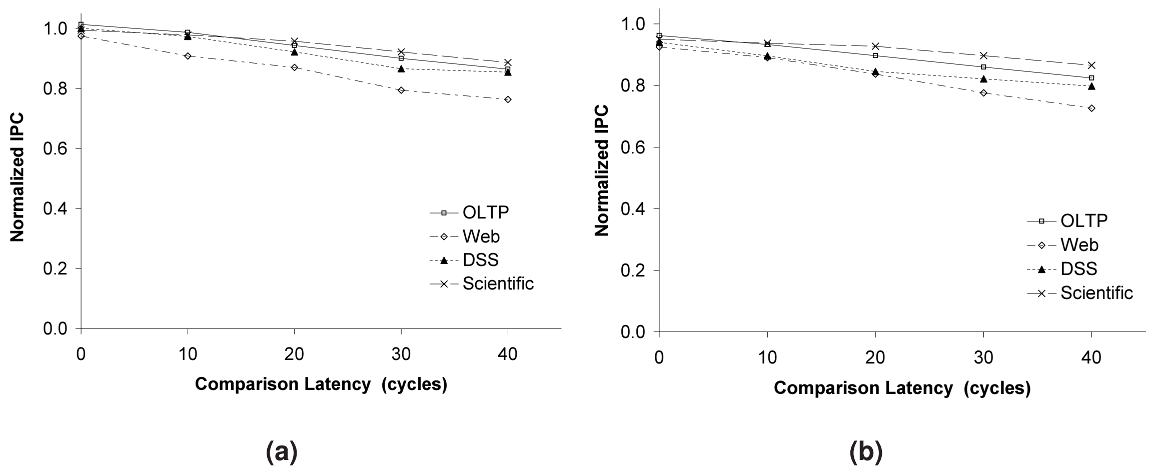


Figure 30: Performance sensitivity to the comparison interval of (a) strict input replication with fingerprint comparison and (b) Reunion's relaxed input replication with fingerprint comparison.

Figure 30(a) shows the average performance impact from checking in Strict for each workload class over a range of on-chip comparison latencies, normalized to a non-redundant baseline. At a zero-cycle comparison latency, the workloads do not show a statistically significant performance difference from non-redundant execution. The performance penalty increases linearly with increasing comparison latency. Both commercial and scientific workloads exhibit similar sensitivity to the comparison latency; however, the mechanisms are different.

In commercial workloads, the dominant performance effect comes from frequent serializing instructions. With increased comparison intervals, the number of these events remains constant, but the stall penalty increases. At forty cycles, the average performance penalty from checking is 17%.

In contrast, the scientific workloads suffer from increased reorder buffer occupancy because they can saturate this resource, which decreases MLP. At a comparison latency of forty cycles, the average performance penalty is 11%. Larger speculation windows (e.g., thousands of instructions, as in checkpointing architectures [4]) completely eliminate the resource occupancy bottleneck, but cannot relieve stalls from serializing instructions.

5.5.3 Reunion Performance

This section now evaluates the performance penalty of relaxed input replication under Reunion and explores Reunion's sensitivity to comparison latencies. Unlike the strict input replication model, vocal and mute execution in Reunion is only loosely coupled across the cores. For non-serializing instructions, these differences can be absorbed by buffering in the check stage. However, serializing instructions expose the loose coupling because neither core can make further progress until the slower core arrives at and compares the instruction. This introduces additional retirement stalls that affect both cores, on top of the comparison overheads discussed above. Figure 30(b) shows the performance of Reunion for a range of comparison latencies. Reunion's performance is determined by checking overheads, loose coupling, and input incoherence.

The first observation from Figure 30(b) is that, unlike Strict, Reunion has a performance penalty from loose coupling and relaxed input replication at a zero-cycle comparison latency. For commercial workloads, the serializing events expose the loose coupling, because one core must wait for its partner to catch up before comparing fingerprints. For scientific workloads, contention at the shared cache increases the effective memory latency, decreasing performance. This result shows that the baseline performance penalty of Reunion's relaxed input replication is small on average, 5% and 6% for commercial and scientific workloads, respectively.

The second observation in Figure 30(b) is that at non-zero comparison latencies, performance converges towards the limits set by the strict input replication model. As the comparison latency grows, the comparison overhead and resource occupancies dominate the performance, because more time is spent waiting on the comparison than resolving loose coupling delays. At a forty-cycle comparison latency, the average performance penalty is 22% and 13% for commercial and scientific workloads, respectively, which closely follows the Strict model's trend. This result shows that the primary performance impact with larger comparison latencies comes from fundamental limits of checking and recovery, instead of relaxed input replication.

Figure 31 shows an execution time breakdown for the lockstep CMP baseline normalized to unity and Reunion, at a 10-cycle fingerprint comparison latency (unlike the normalized IPC graphs, higher bars indicate slower execution in this chart). The differences between the two bars shows the contribution of performance factors for each workload, starting with time where the cores are busy, waiting on memory requests (loads or stores), or serializing atomic or side-effect instructions. Reunion introduces two new categorizes: time spent where the ROB is full and cannot drain because of a pending fingerprint comparison and time spent where execution stalls because of pending fingerprint comparison of serializing instructions.

For the commercial workloads, the primary performance factor comes from increased time spent

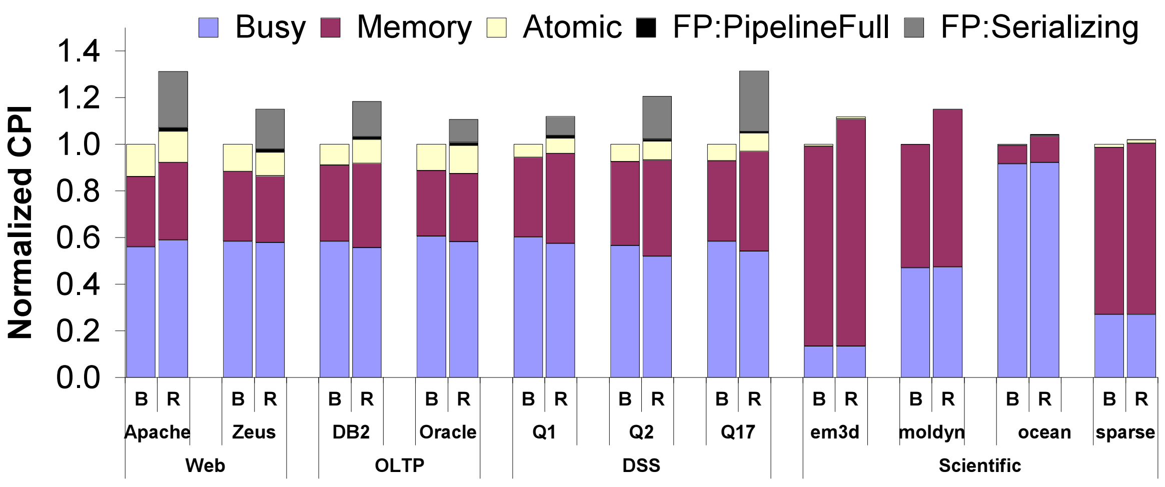


Figure 31. Execution time breakdown for the baseline CMP (B) and Reunion (R).

waiting for fingerprint comparisons during serializing instructions. The fingerprint comparison amplifies the time spent on each atomic instruction. Because the average time spent on each atomic instruction differs across workloads, the relative impact of this comparison also differs across workloads.

For the scientific workloads, the primary performance factor comes from increased time spent on memory operations. Memory latency increases due to vocal-mute contention within the shared cache coherence controller. This is particularly evident for the workloads with high off-chip miss rates: em3d and moldyn. Contention arises because the shared cache controller serializes on-chip and off-chip coherence operations on a cache-block basis. In particular, when a mute request requiring coherence reaches the controller before the vocal request for the same cache block, the cache controller issues an off-chip phantom request for the block. However, because the request is a phantom, it is incoherent and therefore cannot be utilized by the later vocal request. Hence, the vocal request must repeat the off-chip request, in effect doubling the off-chip memory latency when the vocal loses the race. In contrast, when the vocal wins the race, the result will be consumed by the mute through faster on-chip coherence operations.

5.5.4 Input Incoherence

Next, this section provides empirical evidence to demonstrate that input incoherence events in Reunion are uncommon. Table 12 shows the frequency of input incoherence events per million

Table 12: Input incoherence events for each phantom request strength, TLB miss frequency.

	Per 1M instructions				
	Input incoherence				
Workload	**Global**	**Shared**	**Null**	**TLB Misses**	**Atomic Instructions**
Apache Web	0.9	3,818	8,620	1,973	747
Zeus Web	0.2	1,818	5,456	1,654	903
DB2 OLTP	0.7	5,340	16,197	2,492	896
Oracle OLTP	0.6	4,578	17,140	3,297	366
DB2 DSS Q1	21.1	1,909	4,004	207	547
DB2 DSS Q2	0.7	4,852	7,991	1,040	583
DB2 Q17	1.5	4,863	10,466	1,089	833
Avg. Scientific	0.4	17,406	22,607	239	0.22

retired instructions in Reunion for all three phantom request strengths, with a ten-cycle comparison latency. As a point of comparison, the input incoherence events are juxtaposed with another common system event with a comparable performance penalty: the translation lookaside buffer (TLB) miss.

Reunion is first considered with global phantom requests. Recall that global phantom requests initiate on- and off-chip non-coherent reads on behalf of the mute. As shown, the workloads encounter input incoherence events infrequently with global phantom requests, while data and instruction TLB misses generally occur orders of magnitude more frequently. From this data indicates that input incoherence events are, in fact, uncommon in these workloads. Furthermore, even with relaxed input replication, the penalty of recovery is overshadowed by other system events.

Next, this section investigates whether choices for weaker phantom requests from Section 5.4.2 are effective. The data for shared and null phantom requests in Table 12 show that input incoherence events are three to four orders of magnitude more frequent than with global phantom request strengths and in both cases are more frequent than TLB misses. These high frequencies indicate that recovery from input incoherence can become a bottleneck with weaker phantom requests.

Figure 32 compares the performance of the three phantom request strengths with a 10-cycle comparison latency, normalized to the non-redundant baseline. Both shared and null phantom requests incur a severe performance impact from frequent recoveries. Shared phantom requests capture most mute L1 misses because of the high shared-cache hit rate. One notable exception is em3d, whose working set exceeds the shared cache and therefore frequently reads arbitrary data instead of

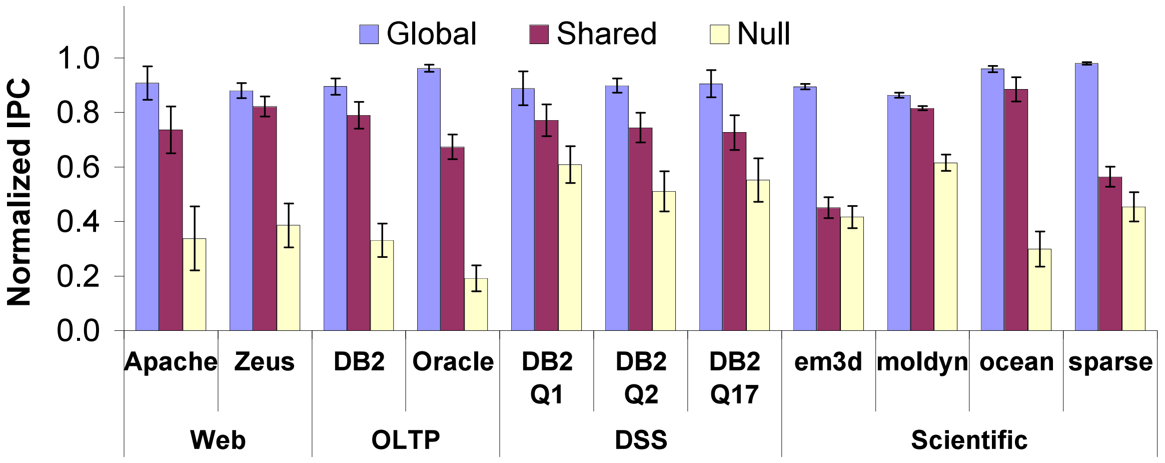


Figure 32: Reunion performance with different phantom request strengths at a 10-cycle comparison latency.

initiating off-chip reads. The null phantom request policy has a severe performance impact for all workloads because each L1 read miss is followed by input incoherence rollbacks.

The analysis of input incoherence in this section shows that global phantom requests are effective in reducing input incoherence events to negligible levels in the workloads. Furthermore, the weaker phantom request strengths increase the frequency of input incoherence to levels that cause a severe performance impact.

5.5.5 Synchronizing request type

Next, this section evaluates the practical performance implications of synchronizing request implementations. Figure 33 shows the performance of both bilateral and null synchronizing requests. There is virtually no sensitivity to the synchronizing request type in any of the workloads [3] This result is expected because, (1) the cost of performing a bilateral (or unilateral) synchronizing request is similar to that of a null synchronizing request, (2) the synchronizing requests are infrequent, and (3) null synchronizing requests are not expected to repeat often. Both request types are dominated by the shared L2 cache access, therefore have similar average access latencies. Synchronizing requests occur infrequently (already demonstrated in Table 12). Finally, the simulation results indicate that *none* of the null synchronizing requests encountered a persistent race, therefore in practice, no null synchronizing requests needed to repeat. This result indicates that simple synchronization re-

[3] A t-test indicates that statistically significant claims cannot be made about the small performance difference in DSS query 1.

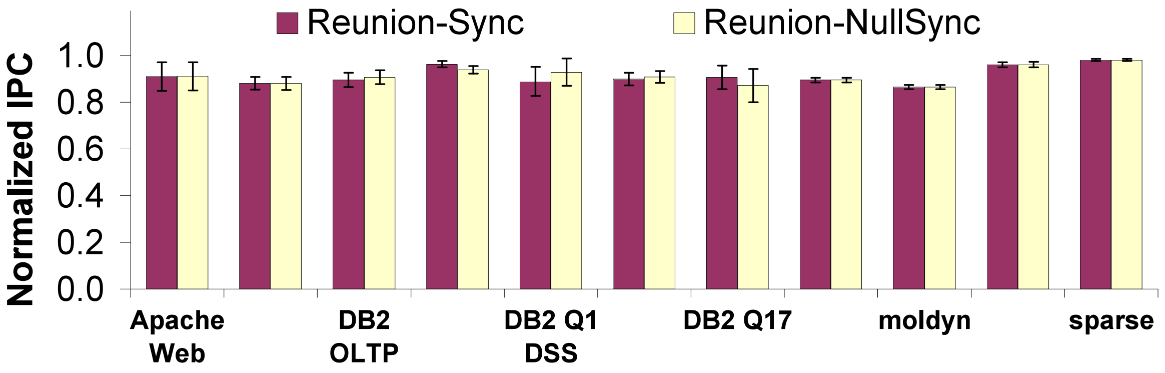


Figure 33: Performance of Reunion with bilateral and null synchronizing requests at a 10-cycle comparison latency.

quest mechanisms are sufficient for good performance on realistic workloads. Furthermore, even if a “hammer” mechanism is needed to guarantee forward progress, it will not be invoked frequently; therefore, it can be simple and slow without incurring a large performance burden.

5.5.6 Serialization Overhead

Next, the importance of architecturally defined serializing instructions to redundant execution performance is identified. In this system, serializing instructions are traps, memory barriers, and non-idempotent memory requests. Many of these events are inherent in the workloads, such as memory barriers needed to protect critical sections, while others, such as system traps, are specified by the instruction set architecture.

The results presented up to this point in the chapter have eliminated the dominant source of system-specific traps in the baseline UltraSPARC III architecture: the software-managed TLB miss handler. In commercial workloads, the fast TLB miss handler is invoked frequently (see Table 12), due to their large instruction and data footprints. The handler function includes two traps, for entry and exit, and executes three non-idempotent memory requests to the memory management unit (MMU).

Figure 34 contrasts the average performance of commercial workloads with a hardware-managed TLB model and the architecturally-defined UltraSPARC III software-managed TLB handler. As the comparison interval increases, the contribution of the serializing checks is readily apparent increasing the performance impact to 28% at a forty-cycle comparison latency. While this result is from Reunion, a comparable impact also occurs with strict input replication.

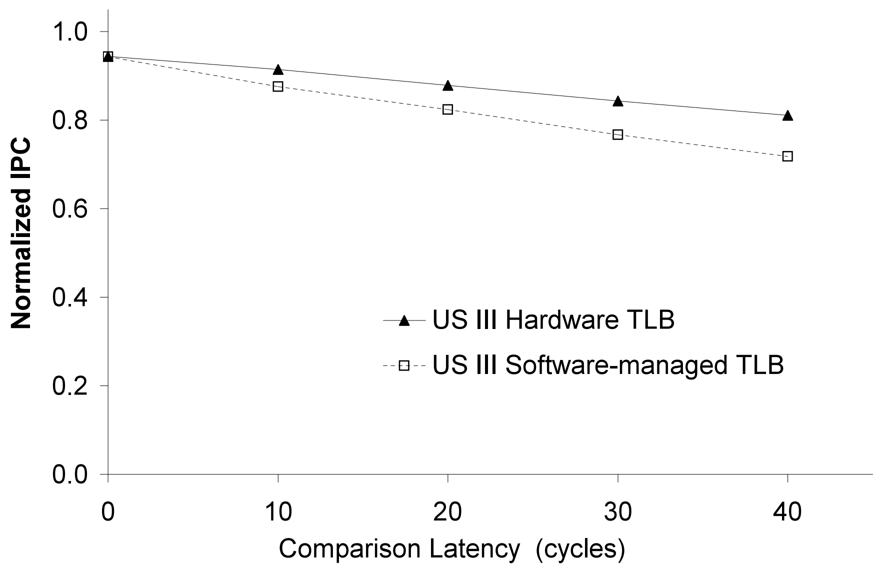


Figure 34: Reunion's average performance for commercial workloads with hardware and software TLBs. Scientific workloads have infrequent TLB misses and are therefore show no sensitivity to TLB implementation.

Strong memory consistency models can also affect checking performance. In contrast with Sun TSO, the Sequential Consistency (SC) memory consistency model places memory barrier semantics on every store (5-10% of the dynamic instructions in the workloads). Hence, every store serializes retirement. An average performance loss of over 60% is observed at 40 cycles due to store serialization with SC.

The results in this section underscore the importance of considering serializing instructions, especially architecture-specific ones, in the performance of redundant execution microarchitectures.

5.5.7 Fingerprinting interval and fingerprints on the interconnect

This section now briefly evaluates the performance impact of moving architectural fingerprint comparison from dedicated channels to the on-chip memory interconnect.

Architectural fingerprints are modeled as an additional cache request that consumes a port on the snoop channel. The snoop channel is a high-priority channel that bypasses the L1 cache and is typically used for acknowledging invalidation and downgrade coherence requests. The fingerprint message traverses the core-l2 cache crossbar twice, with a routing and processing delay at the L2 of four cycles.

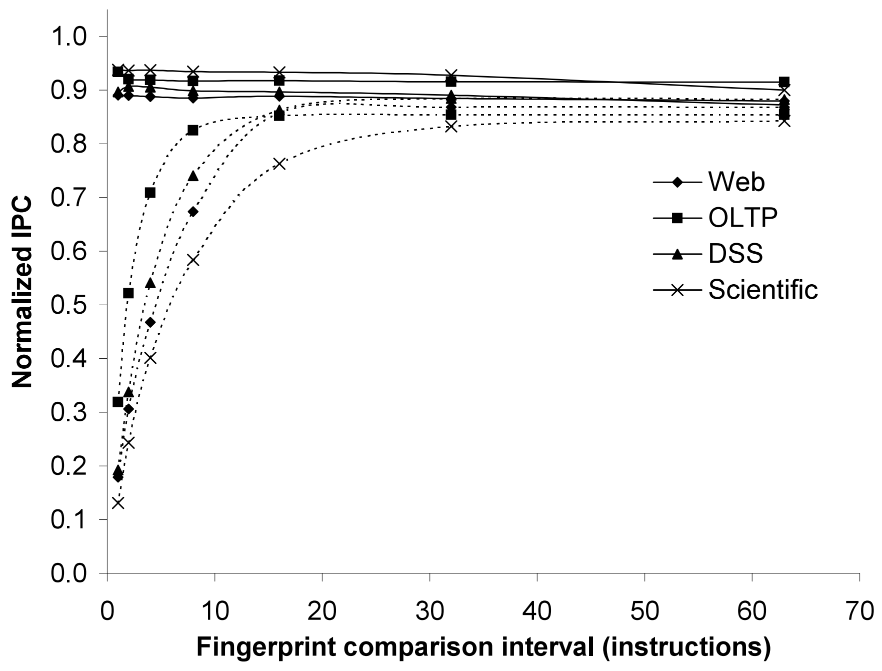


Figure 35: Performance of Reunion as the fingerprint comparison interval is varied for dedicated comparison channels (solid lines) and on-chip memory interconnects (dashed lines) normalized to non-redundant baseline.

Figure 35 shows the performance of Reunion with fingerprints on dedicated channels with a fixed ten-cycle latency (solid lines) and the on-chip interconnect (dashed lines) over a range of fingerprint intervals, normalized to a non-redundant baseline. [4] Workloads have been combined into their respective classes to simplify presentation. The results are equivalent to those presented in [27], however the implementation in this thesis is written independently, the full set of workload checkpoints further tightens the confidence intervals, and the baseline is a non-redundant CMP.

The results show little sensitivity to the fingerprint interval with dedicated interconnects. This result is expected from the both results of the analytical model for the fingerprint interval in Section 5.4.5 and the fact that only the scientific workload Ocean exercises the maximum retirement bandwidth of the four-wide processor core, therefore ample buffering opportunity exists within the ROB. This result shows that the predictions of the analytical model for the fingerprint comparison interval also hold for realistic workloads—there is little sensitivity.

By contrast, the on-chip interconnect shows a drastic (10x) performance loss with a fingerprint

[4]Fingerprint intervals range from a minimum of one fingerprint per instruction to one fingerprint per sixty-three instructions. In this implementation, large numbers of consecutive stores—e.g., more than sixty-four—will cause a deadlock unless the interval is less than or equal to the store buffer size. Such bursts occur in Solaris context switch routines.

comparison on every instruction. This performance loss, relative to a 10-cycle dedicated channel, is almost eliminated as the fingerprint interval increases to sixteen instructions. The scientific workloads show the most significant performance sensitivity with smaller fingerprint intervals, primarily due to cache port contention in ocean, which already has high MLP and IPC near four. The other extreme, the OLTP workloads have low IPCs of 0.3 to 0.5 due to memory stalls. These workloads place an order of magnitude less pressure on the ports and interconnects, and therefore are significantly less affected by contention on cache ports and the memory hierarchy. Compared to dedicated channels at an interval of one instruction, the average L2 cache transaction latency increases by 220%. However, this latency is only 13% at a sixteen-instruction interval and just 4% at a 63-instruction interval. Therefore, the latency impact can be eliminated by choosing the appropriate interval. These results (and further analysis in [27]) show that transmitting architectural fingerprints over the on-chip interconnect is viable—and comparable in performance to a dedicated 10-cycle comparison latency—for both commercial and scientific workloads.

5.6 Conclusion

Designs for redundant execution in multiprocessors must address input replication and output comparison. Strict input replication requires significant changes to the pipeline. This thesis observes that the input incoherence targeted by strict input replication is infrequent. This thesis proposes the Reunion execution model, which uses relaxed input replication, backed by light-weight error recovery within the processor's speculative window. The model preserves the existing memory system, including the coherence protocol and memory consistency model. The results show that the overhead of relaxed input replication in Reunion is only 5% and 6% for commercial and scientific workloads, respectively.

Chapter 6

Microarchitectural Fingerprints

Recent studies have shown that the vast majority of errors do not affect architectural state, therefore looking only at architectural results misses most errors [73, 117]. For faults that worsen with time, such as device wearout, detecting early evidence of their existence (even if architectural state is currently unaffected) is valuable for predicting and tracking future growth. An ideal detection mechanism compares every sequential node with a error-free model on each clock cycle. While manufacturing test approaches this limit by using expensive testers coupled with full-hold scan architectures [55], this solution is impractical for in-field detection where testers are unavailable. As a further constraint, the detection mechanism must (1) avoid false positives from non-determinism in microarchitectural sources such as floating tri-state buses and asynchronous interfaces, both of which carry architecturally-important important outputs, and (2) operate at-speed to detect errors that are timing-dependent.

This chapter introduces the concept of *microarchitectural fingerprints*. Microarchitectural fingerprints detect errors that are visible in microarchitectural state, even if these errors do not propagate to architectural state. In-field detection of emerging device wearout is one application for microarchitectural fingerprints. This application, called FIRST, differs from soft error detection—where it is sufficient to check only retired architectural state. Instead, FIRST requires detecting errors that are architecturally masked during normal operation and therefore do not *currently* affect correct execution but, given time, will develop into errors in architectural state in normal operation.

Because the basic metrics only differ slightly from architectural fingerprints, this chapter is structured assuming an understanding of the discussion in Chapter 2. Section 6.1 introduces the fault model for microarchitectural fingerprints. Section 6.2 describes microarchitectural fingerprints in detail. Section 6.3 explains the metrics as they apply to microarchitectural fingerprints, followed by a hardware design in Section 6.4. The soft error coverage of microarchitectural fingerprints is briefly evaluated in Section 6.5 and then the chapter concludes.

6.1 Fault Model

Microarchitectural fingerprints detect *microarchitectural errors*. Microarchitectural errors are deviations from the implemented microarchitecture—the state in internal latches—of a processor caused by an underlying fault. Microarchitectural fingerprints must are still subject to logical masking, which hides some internal errors. However, because microarchitectural fingerprints inspect internal state that is not always visible architecturally, these fingerprints do not suffer from architectural masking. This thesis does not evaluate latch-window and electrical masking.

The fault model targeted by microarchitectural fingerprints is the same as for architectural fingerprints as defined in Section 2.1 (that is, soft errors and device wearout). However, for practical reasons explained later in this chapter, microarchitectural fingerprints are not well suited for soft error detection applications.

6.2 Microarchitectural Fingerprints

This section introduces microarchitectural fingerprints. Microarchitectural fingerprints capture a reproducible, deterministic hash of internal microarchitectural state during execution to detect differences with respect to a reference execution. Microarchitectural fingerprints leverage built-in observe-only test and debug hardware, known as *signature-mode scanout*, to accumulate a concise summary of microarchitectural state throughout the processor. Scanout consists of chains of special observe-only latch cells added to a design [24].

Traditionally, signature-mode scanout has been of little practical use outside of manufacturing test and debug due the indiscriminate selection of scanout cells within the design which capture non-deterministic values and consequently renders the resulting hash useless. The proposed design provides limited tolerance of non-determinism on monitored nodes, while maintaining coverage of known-deterministic values on those nodes. This design leverages the logic designer's knowledge of the functionality to sample selectively values with scanout only when the values are known to be deterministic. While traditional signature-mode scanout produces a single continuous hash for the entire microprocessor, the proposed design also permits temporal and spatial error localization by collecting the scanout chain output in small (16-bit) linear feedback shift registers (LFSR) or multiple-input shift registers (MISR). This localization information can be useful for applications such as graceful degradation so that error detection can continue after units have already failed in the field [111].

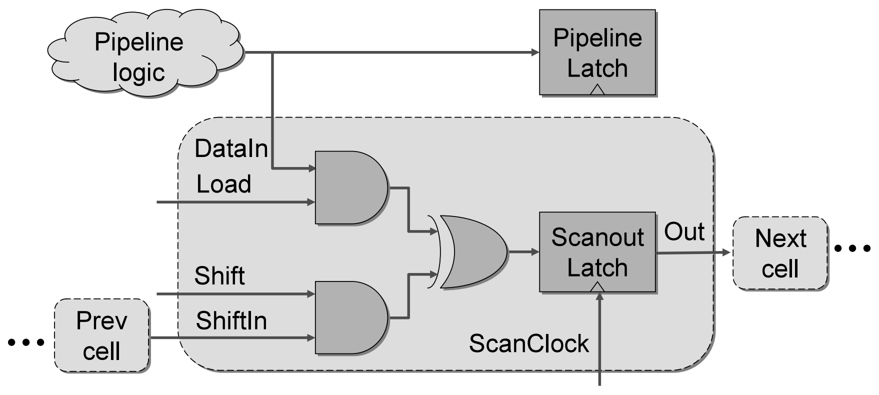
6.3 Metrics

As with architectural fingerprinting, detection latency, comparison bandwidth, and error coverage remain the key evaluation metrics. However, the specific tradeoffs change in this domain.

Detection latency. Detection latency scales with the length of the scanout chains that can easily reach thousands of nodes in industrial designs. However, this latency can be ameliorated by breaking long chains into several smaller chains and combining the resulting chains in parallel. While the detection latency is bounded, the latency may be too long to permit timely recovery with soft error detection. However, because microarchitectural fingerprints are intended to detect emerging errors—not soft error recovery—the detection latency primarily serves to reduce the test time.

Comparison bandwidth. Microarchitectural fingerprints compact data into signatures over space and time. Each chain generates one bit per cycle, which can be fed into a hash circuit. As with architectural fingerprints, the hash is sampled periodically, therefore the comparison bandwidth is primarily a function of the fingerprint comparison interval. Because microarchitectural fingerprints operate below the level of instruction sets and architectural structures, microarchitectural fingerprint comparison intervals are measured in cycles rather than instructions.

Error coverage. Error coverage is limited by logical and electrical masking in the circuit being observed. Furthermore, the location and number of scanout nodes and degree of determinism determine how widely and how often values can be sampled by scanout.



Shift	Load	Operation
0	0	Reset
0	1	Snapshot DataIn
1	0	Shift chain
1	1	DataIn ⊕ ShiftDataIn, shift chain

Figure 36: The scanout cell and its logical operations. Load and Shift are external control signals that allow sampling of the monitored DataIn input and shifting of the ShiftIn input.

6.3.1 Discussion

The long detection latency (thousands of cycles) and determinism requirements can make microarchitectural fingerprints unattractive for runtime soft error detection. Unlike architectural fingerprints, microarchitectural fingerprints do have the restriction that execution must be cycle-for-cycle deterministic. Furthermore, the internal state of the microprocessor must be carefully initialized before collecting a microarchitectural fingerprint. These restrictions limit the instances when microarchitectural fingerprints can be applied—for example, synchronizing replicated inputs between two distant processor cores, and even local units, is increasingly difficult [18, 66]. Therefore, unlike architectural fingerprints, which are expected to operate with arbitrary workloads, microarchitectural fingerprints are realistically limited to executing controlled workloads in deterministic processor test modes [75].

6.4 Hardware Design

Microarchitectural fingerprints are assembled using a combination of scanout cells to monitor individual nodes and LFSRs or MISRs to compress the output over space and time.

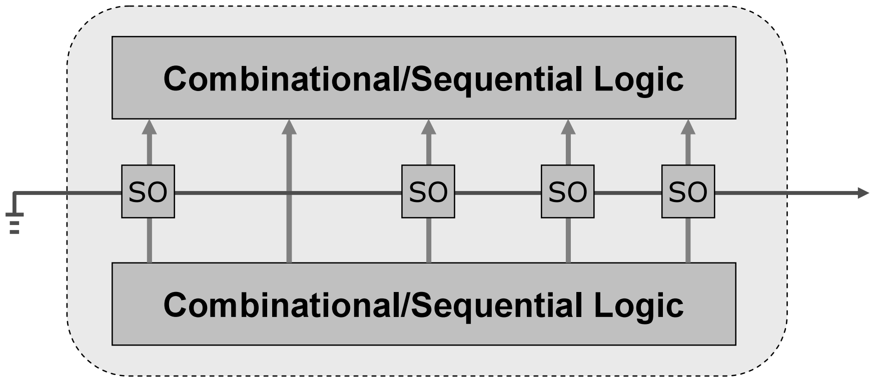

Combinational/Sequential Logic
SO
SO
SO
SO
Combinational/Sequential Logic

Figure 37. Scanout cells applied in a digital circuit.

Scanout Chains. Scanout cells are observe-only latches attached to combinational and sequential nodes in the design that are selected by design engineers who target manufacturing test and silicon debug. The scanout cell and its functions are shown in Figure 36 [24, 55]. The cell can load monitored values into its latch and shift values from a previous cell. In addition, the scanout cell has a "signature mode" operation that performs a logical XOR of previous values in the chain with the currently-monitored value, producing a continuous hash—a fingerprint—of current values combined with prior cycles' state. The scanout cells are linked together as a long scanout chain, similar to traditional scan architectures and as illustrated in Figure 37. However, unlike traditional scan chains, because the scanout chains do not feed into subsequent circuit logic, they can passively monitor the circuit as it operates without changing its intended behavior.

Because designers aggressively add scanout cells to their designs to aid test and debugging, the scanout chains in actual designs commonly include non-deterministic values (referred to as an ”X”) in the scanout signature.[1] Unfortunately, the presence of a single non-deterministic value anywhere along the scanout chain can result in false-positive microarchitectural error detection and the loss of signature-mode scanout coverage for the entire chain. Thus, non-determinism becomes a key factor that limits the usefulness of scanout chains in commercial designs.

In current designs, the non-determinism problem is partially addressed by globally disabling loads on pre-determined cycles or disabling entire chains. However, this masking requires knowing, a priori, the time and location of non-determinism and the masks can only be applied on a limited number of cycles per execution.

[1]Sources of non-determinism include floating values on tri-state buses, uninitialized state following reset, unpredictable external inputs, and marginal timing.

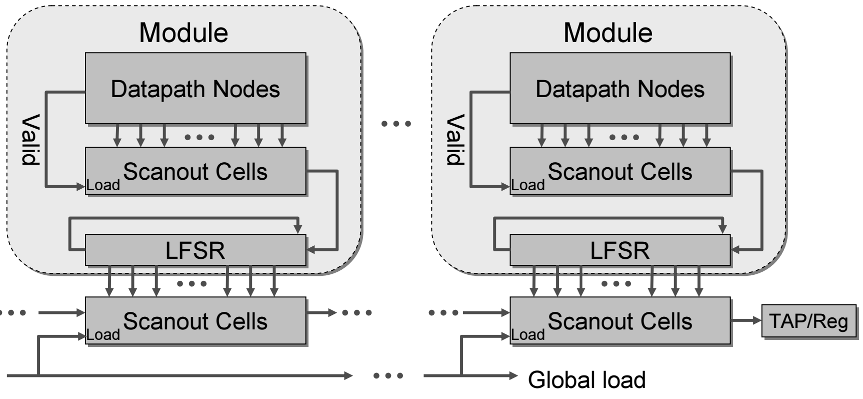


Figure 38: The design of microarchitectural fingerprints: each module contains a scanout chain and local LFSR. The values of each module's LFSRs are sampled and shifted out to internal registers or the test access port (TAP).

Microarchitectural Fingerprints. Microarchitectural fingerprints overcome non-determinism problem of scanout chains, while preserving observability of internal nodes in the circuit. Microarchitectural fingerprints take advantage of the fact that designers have selected nodes for observation and know the node's intended function. This information can be exploited to avoid sampling nodes, on a fine granularity when determinism cannot be guaranteed.

The proposed hardware design for microarchitectural fingerprints is illustrated in Figure 38. Individual scanout nodes can be locally disabled for all or portions of a scanout chain by controlling the ScanLoad signal (the chain still shifts values, independent of the ScanLoad). This allows individual signals on scanout nodes to be dynamically disabled during cycles when the nodes are known or suspected to be non-deterministic. For example, one instance where this is particularly useful is on tri-state nodes: these nodes typically include important datapath values. Furthermore, these nodes can be triggered by existing enable signals from the control path that indicate cycles when the tri-state bus is being driven.

Because microarchitectural fingerprints are intended for in-field error detection, expensive testers—with their associated high external pin bandwidth—will not be available. With traditional testers, full state of the chip can be scanned out for diagnosis of which modules are failing and when, which is important for isolating failures. However, the same techniques cannot be performed in the field. The design for microarchitectural fingerprints preserves spatial and temporal error locations, as shown with the global load signal. Within each module, local LFSRs collect the output of the local scanout chains. These values are periodically transferred out over a global scanout chain to an architecturally-visible debug register or the test access port. At this point, the microarchitectural fingerprint is compared with known-good values for the same execution. This design has the advantage of bounding the microarchitectural error detection latency to the length of the chain and, therefore, the comparison interval. Furthermore, the comparison bandwidth required scales with the number of modules and the hash size, not the total number of bits being monitored.

6.5 Soft Error Injection Evaluation

This section evaluates the coverage of soft errors for microarchitectural fingerprint and compares them to architectural fingerprints in the OpenSPARC T1 RTL model.

6.5.1 Methodology

The statistical error injection experimental framework is the same as described in Chapter 3.4 for architectural fingerprints, except for the addition of a microarchitectural fingerprint model. A custom Verilog PLI module simulates the scanout chains and LFSRs. Microarchitectural fingerprints are applied recursively to all registers in the chosen top-level module, in addition to a boundary scanout chain on the outputs of the module. Bit flips injected into pipeline latches are not registered as microarchitectural errors in the scanout chain unless the erroneous value propagates to a subsequent consuming latch without being logically masked. The microarchitectural fingerprint is compared once after the chain has been fully shifted into an LFSR at the end of each test program execution.

6.5.2 Results

Figure 39 shows the baseline coverage of architectural and microarchitectural fingerprints using the methodology outlined above. Each bar reports the fraction of soft errors injected that were detected as errors by the respective detection mechanism. The individual architectural results are identical to those presented in Chapter 3.4, therefore this discussion is limited to new results with microarchitectural fingerprints and a comparison of the two error detection mechanisms.

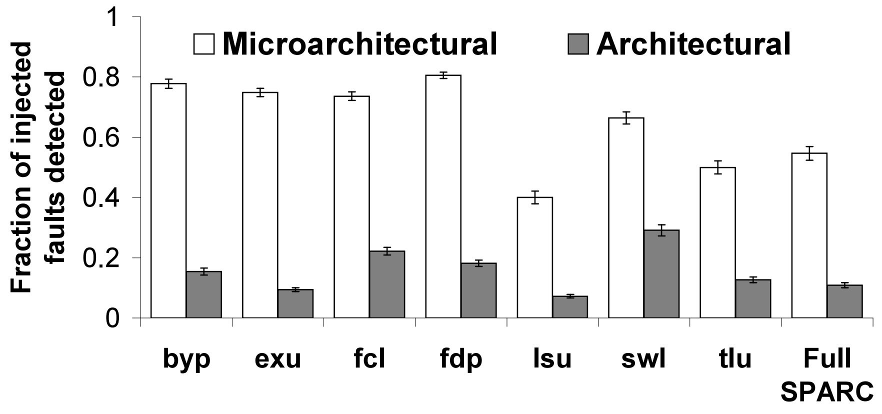


Figure 39. Soft error injection detection results.

Logical masking. Logical masking in the RTL model prevents microarchitectural fingerprints from reaching 100% coverage. This error masking is particularly acute in the load-store unit (`lsu`) and trap logic unit (`tlu`), where complicated control paths and datapaths handle exceptional conditions, but are largely unused during common-case execution. By contrast, the bypass network (`byp`) has a high error detection rate, because it holds operands and results for instructions on every active cycle. Furthermore, the bypass unit is dominated by muxes that select three independent operands from a single set of source values. This design increases the likelihood of propagating the error on at least one path, hence its relatively high fault coverage. Overall, due to the high level of logical masking during program execution, only 55% of the injected errors are detected as errors by microarchitectural fingerprints on the full-core model.

The microarchitectural fingerprint masking is not comparable to the 85% average level of masking at the microarchitecture level reported by Wang et al. [117]. This is because Wang inspects microarchitectural state (all latches) at the end of program execution, where erroneous values can be masked by execution in subsequent cycles. By contrast, microarchitectural fingerprints preserve errors that occur temporarily in the middle of execution, even if they are logically masked in subsequent cycles. Therefore, microarchitectural fingerprints are more sensitive to microarchitectural errors.

Architectural masking. The difference between microarchitectural and architectural fingerprint bars in Figure 39 indicates the contribution of architectural masking in the design. The individual units architecturally mask 77% of the microarchitectural errors, on average, with the thread switch logic being the notable exception. This level is higher than logical masking, where errors need only propagate to a monitored latch.

This result provides motivation against using microarchitectural fingerprints for soft error detection. The demands of soft error tolerance are satisfied if architectural execution remains correct. Therefore, because microarchitectural fingerprints show strong sensitivity to microarchitectural soft errors that never affect architecturally correct execution and the microarchitectural fingerprint provides no indication of whether the microarchitectural error will be masked architecturally, this detection mechanism needlessly detects errors that should be derated [73]. By contrast, for applications where there is temporal correlation between instances of a fault (e.g., emerging wearout detection), detecting architecturally masked errors provides a useful indicator of more widespread future faults.

6.6 Conclusion

In closing, microarchitectural fingerprints provide high observability of the internal microarchitectural state; however, they are limited by the determinism requirement. Microarchitectural fingerprints are further evaluated for emerging wearout detection in Chapter 7.

Chapter 7

FIRST

7.1 Introduction

This chapter introduces a process for early detection of emerging wearout faults in processor cores called Fingerprinting In Reliability and Self-Test (FIRST).

As CMOS feature sizes continue to shrink, transistor and interconnect reliability worsens [65]. While numerous physical phenomena will account for future device failures, the overall system-level impact is shorter and less predictable lifetimes for microprocessors [110].

Unlike traditional manufacturing defect and single-event upset fault models (e.g., stuck-at faults and transient bit flips, respectively), wearout-related faults in future process technologies will appear with gradual onset and will first affect device timing [60, 89]. Designers conservatively add timing and voltage slack—known as a guardband—to ensure that logic meets latch setup times and to provide noise margins. As logic devices transition more slowly over their lifetime, combinational logic paths will eventually fail to meet timing requirements and encroach on the design's guardband [58]. Initially faults will appear intermittent, depending on specific operating conditions (e.g., voltage, temperature, circuit inputs, etc.), but eventually result in permanent failure through hard breakdown. The number of potential critical paths in complex designs—particularly those due to rising within-die and across-die static variations [22]—makes the task of predicting the first paths to develop wearout difficult.

Recent work advocates detecting and recovering from errors caused by wearout as they occur during normal operation. This thesis refers to such techniques as *just-in-time error detection*. Prior proposals integrate carefully designed error detection mechanisms into existing designs, but face significant challenges from the increasing number of unpredictable critical paths [19], need custom latch designs [2, 33] or require integration with resource scheduling mechanisms [98]. Bower et al. [23] avoid the challenges of circuit-level test, but require extensive design changes to support instruction-level checking.

This thesis proposes *early error detection* with FIRST: Fingerprinting in Reliability and Self Test. FIRST uses infrequent, periodic (e.g., once daily) testing where application and system software are suspended from a core, and the core is subjected to marginal operating conditions while running special test programs to detect the onset of wearout. FIRST reduces the processor's effective guardbands to expose marginal critical paths well before developing wearout faults affect normal operation on those paths.

This thesis evaluates detection two mechanisms for use with FIRST: microarchitectural and architectural fingerprints. Microarchitectural fingerprints provide a lightweight signature of microarchitectural state updates using existing test hardware, while architectural fingerprints summarize architectural state updates with dedicated hardware added at the retirement stages. Both techniques allow detection of growing wearout faults across the processor core without requiring advance knowledge of which devices will fail.

The contributions of this work are:

- **FIRST:** This thesis introduces the idea of reducing guardbands to provide early runtime detection of device wearout.

- **Wearout fault model:** This thesis presents a path delay fault-based model of wearout that is fast enough to simulate unit-wide faults, with FIRST, on full-chip RTL.

- **Detection mechanism evaluation:** This thesis evaluates coverage of wearout detection mechanisms over a range of representative units in full-chip RTL simulation using the FIRST procedure. The results show that microarchitectural fingerprints provide high coverage for isolated wearout, while architectural fingerprints are as powerful as microarchitectural fingerprints for detecting widespread development of wearout.

This chapter is organized as follows. Section 7.2 provides background on wearout faults. Section 7.3 introduces the FIRST concept and implementation. Section 7.4 presents a simulation model for wearout faults. This thesis evaluates the simulation model and FIRST procedure with fingerprints in Section 7.5 and then concludes.

7.2 Background

Sources of device wearout include gate oxide breakdown, hot-carrier injection, negative-bias temperature instability (NBTI), and electromigration [65]. The onset and end stages of device wearout (soft breakdown and hard breakdown, respectively) have been studied extensively in device reliability literature using accelerated wearout testing techniques. The effects of device wearout are expected to worsen with process technology scaling [65].

During gate oxide soft breakdown, transistor switching speed decreases for a given operating voltage [58]. The logical operation of the transistor is nevertheless maintained [11]. This behavior means logic gate outputs transition more slowly as the soft breakdown progresses. Similar results have been shown for NBTI [87] and hot-carrier injection [28].

In light of this behavior, wearout manifests as slow rising or falling transitions on the affected devices [25]. For example, a NAND gate with a failing NMOS transistor experiences a slowdown for the falling output, while a failing PMOS transistor experiences a slow rising output transition. Depending on the precise fault location in a logic gate, speedups are also possible [89], however slow-downs still dominate performance and are decidedly detectable.

While today's designs have multiple statically known critical paths, increasing within-die variation associated with process scaling means that a particular die's critical paths are not necessarily known at design time [21]. Furthermore, experiments show that switching speed decreases dramatically during soft breakdown [87], which means that existing critical paths lengthen and new critical paths can arise. While growing levels of process variation mean that some devices are inherently more likely to fail than others, the development of wearout faults is also strongly temperature and voltage-dependent [19, 65, 109, 110] and therefore *widespread* instances of faults can develop concurrently within the same unit. This is shown in simulation-based studies of NBTI [78] and gate oxide breakdown [19]. Because of these issues, predicting the location and number of wearout faults is difficult and detection mechanisms must look broadly across the design to detect the timing changes.

Although device wearout is gradual and the precise failure locations are unpredictable, the failure rate is shown to fit time-dependent distributions (e.g., log-normal and Weibull [58, 111]) and the failure rate is known to increase with time. Furthermore, empirical evidence shows that once a device has begun the failure process, the degradation rate increase with time and is also strongly affected by operating conditions (e.g., exponentially related to supply voltage) [60]. With typical operating conditions, the soft breakdown occurs gradually and progressively over days or weeks. Thus, there is opportunity for early detection of devices that begin soft breakdown.

7.3 Detection with FIRST

Based on the observation that common wearout faults exhibit a gradual onset, this thesis proposes the FIRST (Fingerprinting In Reliability and Self Test) methodology for early detection of device wearout. The key idea behind FIRST is to test periodically the processor core in near-marginal conditions to expose changes in timing that initially hide inside the processor's frequency and voltage guardbands. FIRST can utilize both architectural fingerprints (described in Chapter 2.2) and microarchitectural fingerprints (described in Chapter 6) to compact and efficiently compare the outcome of each test.

Effective early wearout detection demands (1) mechanisms to induce marginal operation that exposes wearout faults in the processor core and (2) extensive coverage of circuit nodes to detect the developing marginal faults.

The FIRST procedure performs in-field wearout detection in processor cores. To start a test period, the operating system temporarily takes the core offline and places it in a deterministic test mode. The core then loads deterministic functional test programs that exercise logic transitions within control logic and datapaths. While the programs execute, the core generates an at-speed fingerprint of execution that summarizes internal microarchitectural or architectural state updates. The core repeats the same programs and fingerprint collection as operating conditions are gradually moved to less-conservative operating corners (e.g., by lowering supply voltage or increasing clock frequency) which effectively reduces the frequency and voltage guardbands.

When the recorded fingerprint no longer matches those of earlier executions with more conservative conditions, the frequency guardband has been exceeded and the test period ends. A fingerprint mismatch at progressively more conservative operating conditions in subsequent test periods (e.g., over several days) indicates the onset of wearout. The appearance of a small number of faults presages extensive future failures from hard breakdown and severe soft breakdown. The early warning allows time for scheduled replacement or removal of the failing processor.

The fingerprint for a suite of tests programs and information on when they begin to differ can be stored over time by the operating system or in the BIOS using machine check storage [82]. The long-term storage and analysis of this information is beyond the scope of this work; however, trending algorithms such as those utilized by Blome et al [19] can potentially be applied to identify and track the onset of wearout or extend the lifetime of the processor.

7.3.1 Inducing Marginal Operation

Several knobs are already available for artificially producing a near-marginal operating environment in the processor core, including voltage regulators with dynamic voltage scaling, dynamic clock frequency and width controls, and thermal monitoring and control [37]. These mechanisms are discussed below.

Dynamic voltage scaling. Modern processors request changes in the voltage regulator output to save power [46]. Lower voltages cause slower switching speeds that also serve to reduce the guardbands for a given clock frequency. Wearout faults have been empirically shown to increase frequency sensitivity to operating voltages [87]. This mechanism provides a safe and practical procedure for inducing marginal operation.

Dynamic frequency scaling. Processors also include clock frequency scaling capabilities to reduce power consumption during periods when the processor is idle. This capability can be employed to decrease the core's clock cycle time and consequently reduce the guardband. Some processors also allow regional adjustment of clock skew and temporary phase shrinking and stretching [92], which can isolate sections of the design for testing. Test modes also provide deterministic operation of the processor core, although sometimes only at a subset of possible clock rates [113]. This mechanism provides a way to reduce directly the frequency guardbands, but the mechanism may be constrained by the design's power envelope.

Thermal scaling. On the monitoring side, processors contain thermal diodes to measure temperatures across the die. Coupled with functional test programs that run power-consuming instruction sequences, the diodes can help establish a desired core temperature. Given the well-known relationship between switching speeds and temperature in CMOS, this factor also temporarily reduces the guardband and exposes wearout faults. As with dynamic voltage scaling, this mechanism indirectly reduces the frequency guardband, but as with dynamic frequency scaling, may also be constrained by the design's power envelope.

While all three methods discussed in this section will reduce the guardband, dynamic voltage scaling is the safest procedure for long-term reliability. Unlike dynamic voltage scaling, dynamic frequency and thermal scaling both have the potential to place extra thermal stress on the processor, which accelerates wearout.

7.4 Wearout Fault Modeling

A key challenge in evaluating microarchitecture-level device wearout detection methodologies such as FIRST is accurately modeling the effects of wearout faults. This section outlines the wearout fault injection study and simulation framework.

7.4.1 Wearout Fault Injection Study

The overall goal is to understand the fault coverage characteristics of a range of in-field wearout fault mechanisms. Detection mechanisms range from simply checking the output of functional test programs to detailed observation of internal microarchitectural state.

Integral to the study of wearout faults is a model of wearout's effect on logic. This thesis models wearout faults as path delay faults [102], a model that is used in manufacturing tests to exercise critical paths [74]. Because wearout faults appear as increased switching times, the path delay model applies to modeling the effects of wearout. A path is a sequence starting at a primary input (the output of a latch), through a sequence of cells, to a primary output (the input of a latch).

The path's delay determines how long a transition on the input of the path takes to propagate to the output. When the delay along the path exceeds the clock cycle time, the values in the primary outputs may be latched incorrectly, resulting in errors that appear as bit flips in digital circuits.

This model allows investigation of wearout faults in a baseline circuit with pre-determined path delays. While this represents a static view of a circuit's path delays, the simulation environment controls the clock period to model the application of FIRST to a processor with a fixed degree of wearout. The framework is extended to inject errors due to wearout into specific paths in the microprocessor logic. This framework allows investigation of wearout detection techniques over different instances of wearout.

7.4.2 Wearout Fault Simulation

The path delay values are easily generated from synthesis and can be simulated efficiently in an RTL model. In this test, the circuit's critical paths are identified and logic transitions are monitored for triggering conditions. On a matching transition during gate-level simulation, the affected logic is forced to an erroneous value to model the activated fault.

To maintain reasonable simulation speed and to support a large number of simulations, faults should be simulated in RTL (register-transfer level) instead of gate-level models. However, RTL generally models combinational and sequential logic without accounting for delay. Given wearout's similarities to path delay faults, delay fault simulation should provide an accurate model of wearout faults for coverage evaluation. A technique similar to that used in SpeedGrade [50] is implemented to achieve the accuracy of gate-level simulations for path delay faults, but with simulation of RTL and wearout fault trigger conditions.

Fault Simulator Wearout faults are modeled using the flow shown in Figure 40. The input is an RTL description of the circuit to be analyzed, a test program, and a selected clock frequency that determines the slack in the guardband. The output consists of activation statistics for wearout fault sites and error coverage of the detection mechanism.

From an RTL description, ASIC synthesis with a standard cell library generates a list of path delays in the circuit. The path delay list contains timing estimates for both rising and falling input transitions, based upon the standard cell library's characterization for each pair of primary inputs

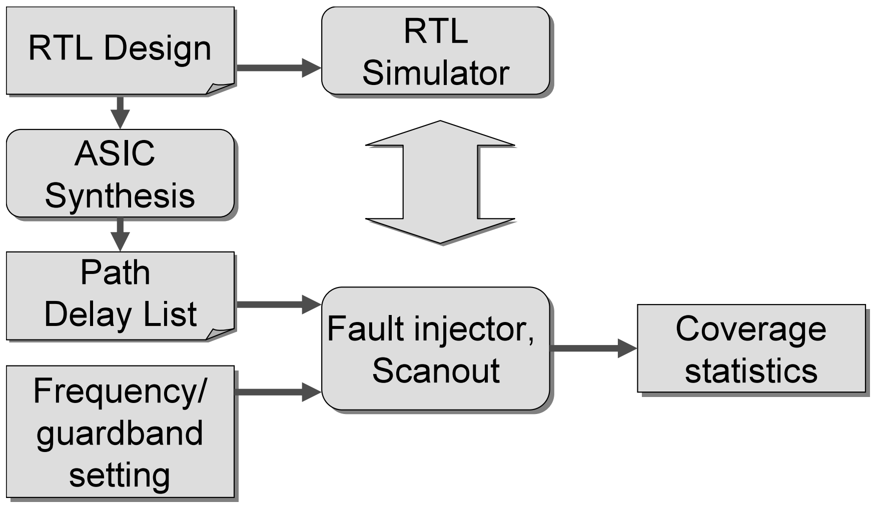


Figure 40. The tool flow for modeling wearout faults.

and outputs that form a path. This information is used to determine when a particular path is a candidate for missing timing in a baseline model of the circuit without wearout. For each circuit, the path delay list is static and only needs to be computed once. The path delay list's delays can be altered later to model the onset of wearout in a once-good unit.

Next, an RTL simulator is augmented with a custom wearout fault injection and scanout chain model. The RTL simulator is a commercial Verilog simulator, while the fault simulator and scanout chains are C-language based libraries that communicate with the simulator through the Verilog programming language interface (PLI). The fault simulator works as follows. The path delay list is read for the circuit. The selected guardband, specified as a clock period in this model, is used to eliminate paths from consideration in later simulation. If it can be determined that the selected clock period meets timing for all possible input transitions on a path, the path is eliminated from consideration.

On each simulated clock cycle, the fault simulator identifies transitions on primary inputs that *potentially* cause the primary output to miss timing. If timing is missed, the combinational logic driving the primary output should behave as if the primary input had never transitioned (i.e., stuck at the previous cycle's value). In this situation, the primary output determined by the RTL simulator needs to be recomputed with respect to the fault model.

This process, however, is not as simple as flipping the output bits, however. Because the primary input may not be a controlling value, the primary output may *still* be unaffected by the path delay fault. To determine the correct values for primary outputs, the fault simulator temporarily rolls back the input transition (by forcing the previous cycle's value on that node) and steps the Verilog model to update the corresponding primary outputs. The updated primary outputs are then compared with the original fault-free output to determine whether the error was masked by a controlling input or propagated to the primary output. The simulator then restores the original primary inputs for unaffected paths. If the affected primary output does differ, the simulator forces the primary output to its erroneous value. The simulator then advances the clock. The fault simulator also collects statistics for each path, including tracking whether any triggering conditions occurred and whether the induced errors propagate to a primary output or are masked.

Instances of wearout are modeled by applying a distribution of added delay along paths in a path delay model for selected units. In this study, a uniform random delay of up to 10% is added to the original delays (estimated by synthesis) across all paths in the selected unit. This process is repeated with different random seeds to model different wearout patterns that can occur in each processor core over its lifetime. Averages over all of these instances are reported to estimate the behaviors over a population of processors affected by wearout.

7.5 Evaluation

This section characterizes the wearout fault model and evaluates FIRST's detection capabilities using architectural and microarchitectural fingerprints. The objectives of these experiments are to demonstrate and compare the detection capabilities of architectural and microarchitectural fingerprints for the early stages of wearout.

This evaluation is structured as follows: a preliminary study shows the baseline feasibility of FIRST using microarchitectural fingerprints. Next, architectural fingerprints and microarchitectural fingerprints are compared for widespread wearout across the unit; these results are further analyzed to identify instances where silent data corruption resulted. Finally, the detection mechanisms are compared for a single instance of a wearout fault.

The wearout fault modeling tool flow from Section 7.4 is applied to several representative units of the Sun OpenSPARC T1 release 1.4 [113] with architectural and microarchitectural fingerprints added, as described in Chapters 2 and 6. The OpenSPARC T1 is a multi-threaded, multi-core chip design, made available in synthesizable Verilog RTL. The evaluated units are listed in Table 13 and their functions are briefly described here. The units are selected to evaluate a range of representative circuit types in a typical processor core. Note that the synthesis does not count SRAMs and supporting logic (due to a lack of a memory compiler), and full path counts and maximum paths across modules are unavailable for the full core model in this synthesis flow. Finally, the individual units represent a sampling of the entire core; therefore, the sum of the measures for each unit does not match the total for the core.

- The execute bypass (`byp`) logic is pure datapath circuitry which contains the operating forwarding and bypass network for the ALU and load values, as well as ECC computation for results being written to the integer register file. The critical paths are clustered in the pipelined ECC generation logic.

- The execute ALU (`exu_alu`) unit contains the datapath for the pipeline's 64-bit ALU and logical operations. The critical path, however, is in error detection for invalid virtual addresses. Because the test programs used to evaluate FIRST always use valid virtual addresses, the effective critical path for this study is on the bus lines between datapath operand inputs and operand outputs.

- The fetch control logic (`fcl`) contains the control path for the instruction cache, fetch stage, and program counter/next program counter management for the entire pipeline. The critical path is in trap/exception determination for fetching the next cycle's instruction.

- The fetch datapath (`fdp`) is the datapath controlled by `fcl`. It contains fetched instruction bits, program counters for the entire pipeline, as well as PC+4 and PC+branch offset computations (48-bit adders), similar to the `exu_alu`. The critical path is in the next fetch PC selection and computation.

- Finally, the thread select logic (`swl`) contains a set of finite state machines that control the ready-to-execute state across four threads and selects one thread to issue on each cycle, based upon readiness, fairness, and pipeline structural hazards. The critical path is in the next state selection.

Table 13: Structural information for the studied microarchitectural units studied in the OpenSPARC. These measures include the small additional logic required to integrate architectural fingerprints into the pipeline.

Unit Name	Registers	Latch Bits	Area (μ^2)	Standard Cells	Paths	Maximum path (ps)
Execute bypass (`byp`)	110	708	141,767	5,450	19,783	769
Execute ALU (`exu_alu`)	377	2,198	48,881	1,734	26,887	1,337
Fetch control path (`fcl`)	135	280	51,033	2,698	11,323	826
Fetch data path (`fdp`)	31	1,358	177,077	8,463	41,210	800
Thread select logic (`swl`)	80	190	39,733	2,193	7,061	993
Full core	2,827	22,095	3,209,188	127,325	—	—

Due to the lack of a memory compiler, timing estimates for SRAM-based structures such as the register file or TLB cannot be determined; however, the datapath portions of these structures are similar in nature to the bypass network.

7.5.1 Feasibility of FIRST

This thesis first evaluates the potential of microarchitectural fingerprints for detecting the timing faults that are crucial to the effectiveness of FIRST.

The path delays are calculated using the Synopsys Design Compiler mapping to the Artisan/TSMC 0.18um low-power standard cell library [9]. In this experiment, the statistics differ from Table 13 because this evaluated unit lacks architectural fingerprint support (this difference does not affect the conclusions of this experiment). The longest transition delay in the unit is estimated to be 951ps over 6,929 paths between 186 latch bits. The unmodified thread switch logic is simulated using Synopsys VCS. The scanout chains are modeled using Verilog PLI and accumulated in a 16-bit LFSR. The circuit is exercised with uniform random input vectors at the module level for 10,000 input vectors following a reset sequence (additional input vectors do not significantly change the observed results) and a cool-down time to shift remaining values out of the scanout chain. The modeled operating frequency is varied between 900ps and 955ps. The baseline circuit delay, without wearout, is employed for this result.

The fault activation results are summarized in Table 14. The total number of potentially sensitized paths is listed in the second column. As discussed in Section 7.4, path delays that are shorter than the clock period are discarded from consideration; therefore, the number of sensitized paths

Table 14. Fault activation results for the thread scheduler over a range of clock periods.

Clock Period (ps)	Possible Paths	Activated Paths	Propagated Paths
955	0	0	0
950	4	1	0
945	9	1	0
940	14	2	0
935	22	2	0
930	33	3	0
925	51	6	1
920	73	12	4
915	99	16	4
910	137	20	6
905	181	25	7
900	207	28	9

increases with shorter clock periods. The third column indicates the activated paths. Activation occurs when there is a transition at the primary input of a path between two consecutive input vectors. For most paths in this circuit, no transitions are observed at the primary inputs that activate the fault. Furthermore, because of logical masking even fewer of activated paths sensitize and propagate an incorrect value at the primary output (shown in the final column). The baseline circuit has 6,929 total paths, of which 2,459 and 1,029 can be activated and propagated, respectively. As shown in the table, the vast majority of these paths are shorter than 900ps. One important result of this experiment is that the statically determined longest paths do not necessarily determine the minimum usable clock period, even if activating transitions can be produced at primary inputs.

The final column in Table 14 indicates when timing faults manifest as microarchitectural errors. The first such failure occurs at 925ps. The microarchitectural fingerprints accumulated over each test period correctly distinguished executions where all timing has been met (over 925ps) from those where an error had been propagated to a primary output (925ps and below). That is, as the clock period is decreased, the tested fingerprint begins to mismatch with the error-free fingerprint at 925ps, just as the first error propagates to a latch.

This result shows that microarchitectural fingerprints differ even when only a single path delay fault has caused a path delay error during the test. The microarchitectural fingerprints clearly

Table 15. OpenSPARC processor parameters.

Core	Single 64-bit 6-stage scalar pipeline
	4-way hardware multithreaded
	1 ALU, 3 read, 2 write port ECC-protected RegFile
	1 load, 1 store cache ports
	1GHz core frequency
L1 Cache	16KB I-cache, parity-protected
	8-KB D-cache, write-through, parity-protected
L2 Cache	3MB 12-way associative, 4-banks
	Unified, ECC-protected, 7-cycle hit latency
Memory	256MB 400MHz DDR2 DRAM
	128 cycle latency

identify when one or a small number of paths in the design begin to miss timing. Therefore, microarchitectural fingerprints do provide the high fault observability needed for the FIRST methodology.

7.5.2 Wearout Detection with FIRST

The previous section established that microarchitectural fingerprints are clearly sufficient for detecting errors from timing faults in the thread switch logic. The next question is whether architectural fingerprints on a processor core running test programs are also sufficient for error detection with device wearout.

Methodology

This section describes the methodology for comparing detection mechanisms using validation test programs running on a single processor core. The processor parameters match those of the shipping Sun Niagara T1 processor and are summarized in Table 15.

Because the number of combinations of faults and processor states is enormous, a sampling methodology is employed to estimate the behavior of different detection mechanisms over a range of wearout scenarios and test programs. Wearout faults are simulated in the baseline path timing for each unit to produce sixty-four instances of wearout, as described in Section 7.4 (each instance starts with a different random seed and adjusts the delay of all paths in the unit). Error injection is enabled as the processor enters the `main` function of each test program, following execution of reset and initialization code.

Table 16. The test programs used to evaluate FIRST.

Name	Dynamic instructions	Test Description
`dram_mt_4th_loads _attrib_many`	3,610	DRAM load/store misses
`exu_irf_local`	39,538	Local windowed registers and bypass network
`mt_alu_ldx`	1,264	Combination of ALU, load, and endian programs
`mtblkldst_loop`	2,564	Back-to-back block loads/stores
`mt_Ifill_L2`	1,582	I-cache fills/misses
`mt_raw`	2,018	Combination of read-after-write programs
`tr_tixcc0`	4,232	Integer condition code traps

Each of these wearout instances is executed with seven multithreaded validation test programs supplied with OpenSPARC T1. These programs, summarized in Table 16, are selected to test a range of units in the processor. The test programs are written for architectural verification. While they focus on exposing improper operation architecturally, the test programs are not specifically written to activate and propagate all possible errors to architectural state. Therefore, with more targeted test programs, wearout faults could potentially be activated at lower degrees of stress and additional faults can be directed towards architectural state. In this situation, architectural fingerprints can improve their coverage, approaching that of microarchitectural fingerprints.

For executions where all threads of the program complete, the comparison of fingerprints with a error-free model is reported after the last thread completes and the scanout chains have shifted out completely. Microarchitectural fingerprints are gathered on all latches within the unit, as well as a boundary scan on all outputs of the specified unit. Architectural fingerprints are gathered on all architectural register and store value/address retirements, as described in Chapter 3. For executions where the processor ceases executing on at least one thread, the results of fingerprint comparison (both architectural and microarchitectural) are reported following a timeout of 10,000 clock cycles of inactivity, which is well beyond the maximum latency experienced by normal on-chip or off-chip operations. The timeout condition, which is a trivial—but sometimes effective—detection mechanism, is also reported separately.

Baseline wearout error detection

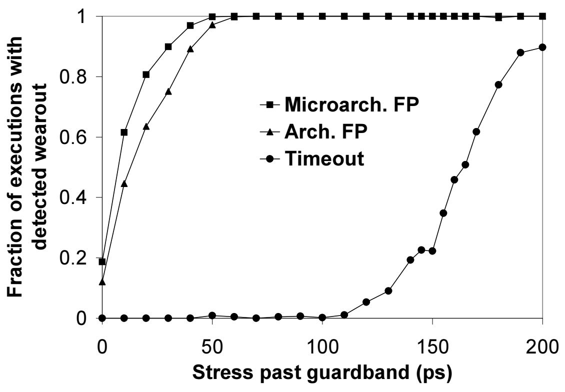
The detection coverage is evaluated for three error detection mechanisms: microarchitectural fingerprints, architectural fingerprints, and a simple timeout.

Figure 41 reports the results for the test programs, averaged over a range of wearout conditions. Along the horizontal axis, the processor is stressed by decreasing the clock period. The level of stress ranges from zero, where the clock period is equal to the longest path in the unit, up to 200ps. The vertical axis indicates the fraction of executions in which an error was detected by each detection mechanism. This measure is the coverage of the detection mechanism over these test programs and faults. This fraction is out of 448 runs per clock period for each unit (sixty-four instances of wearout and seven test programs).

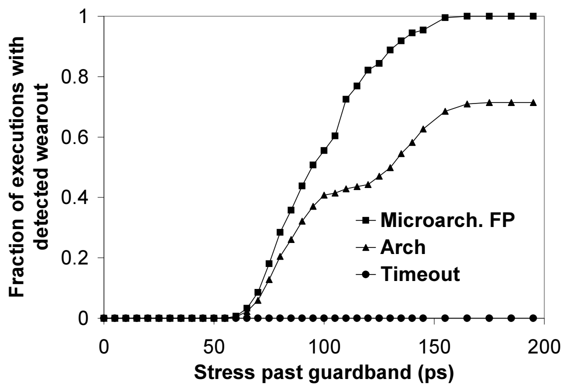
In each plot, the microarchitectural fingerprint curve also establishes the bound for "perfect" error detection. Microarchitectural fingerprints have high observability of errors propagating to a latch. In practice, these fingerprints show the same behavior as perfect detection, modulo the extremely rare instance of aliasing. This behavior has been verified in the simulator, where software can distinguish between masked and true activations (errors) for each fault.

The microarchitectural fingerprint curve is not constant at unity because, although path delay faults can be activated at a longer clock period, they can also be logically masked and therefore produce the same fingerprint as an error-free execution, as demonstrated in the previous section. Hence, the curve should monotonically increase from zero as additional faults activate at higher stress levels. Furthermore, other detection mechanisms, with imperfect coverage, will fall below—or at best, match—the microarchitectural fingerprint curve.

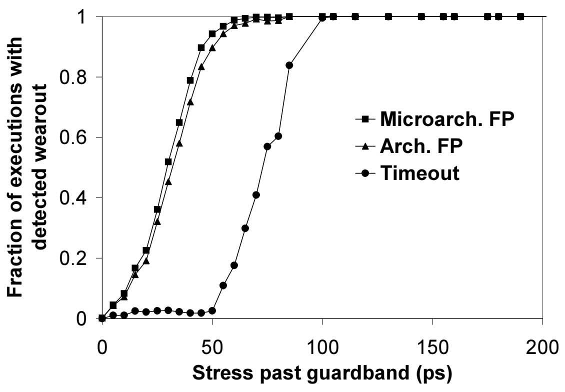
The high-order result to observe from each of these graphs is that first, with the exception of `exu_alu` unit, the architectural fingerprint curve maintains only a small gap between itself and the microarchitectural fingerprint curve. Second, the architectural fingerprints quickly converge to the high level of coverage of microarchitectural fingerprints. This result indicates that architectural fingerprints are as effective as microarchitectural fingerprints in detecting wearout in the FIRST framework. That is, for the same level of stress, architectural fingerprints are as effective as microarchitectural fingerprints in detecting wearout.



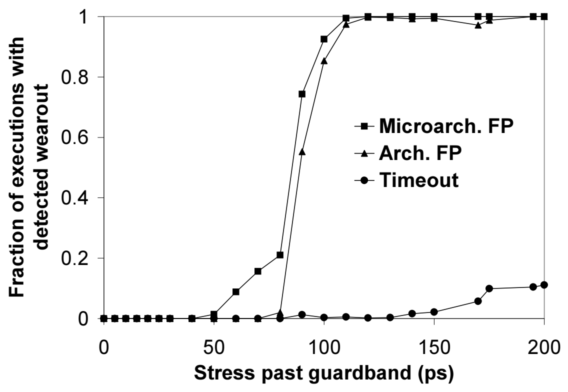
(a) byp



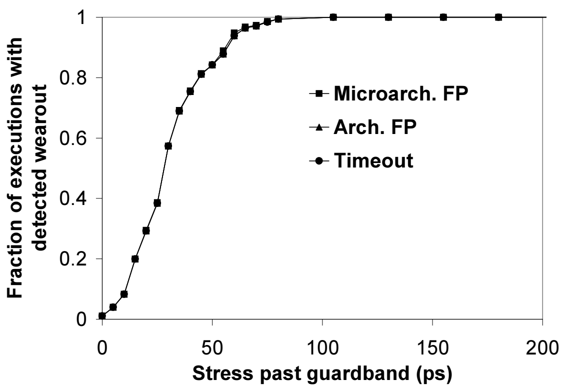
(b) exu_alu



(c) fcl



(d) fdp



(e) swl

Figure 41: Baseline wearout fault detection coverage for microarchitectural fingerprints, architectural fingerprints, and timeout mechanisms. As each unit's clock period is reduced, the number of wearout fault paths increases. Coverage is measured as the fraction of total executions where the mechanism detected an error.

Next, this thesis examines each unit's results in detail. For the execute bypass unit in Figure 41(a), wearout directly corrupts architectural state values within the bypass network. Archi-

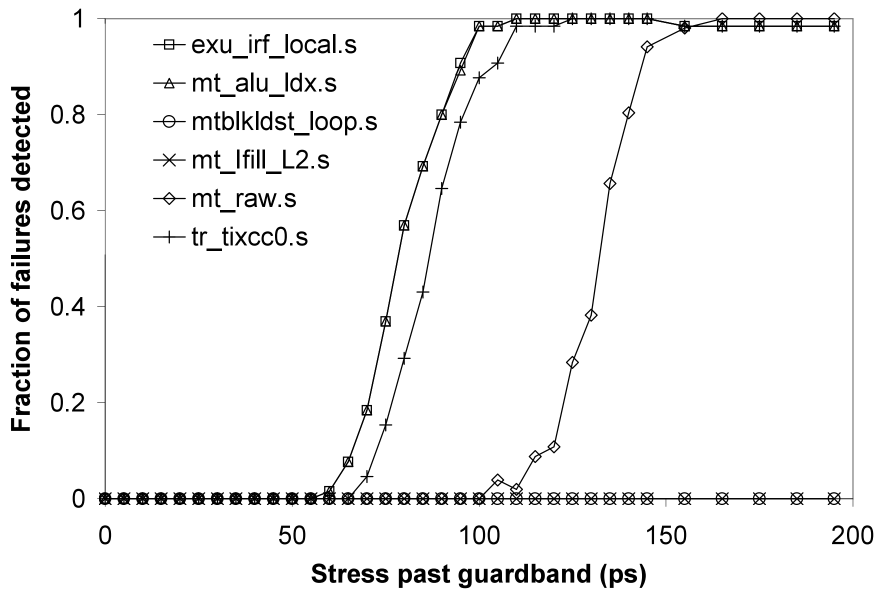


Figure 42: Wearout fault coverage for architectural fingerprints and the exu_alu unit, separated by test program. The mtblkldst_loop and mt_lfill_L2 fail to activate any true faults in this unit, while the remaining programs activate true wearout faults at varying levels of stress.

tectural fingerprints capture virtually all detectable errors with a low level of stress. This will be further analyzed in the following section. The lack of timeouts until high levels of stress also indicates that the processor continues to run the test programs with silent data corruption effects, despite the injected errors.

Similar results are shown for the execute ALU unit in Figure 41(b). Timeouts are simply not triggered at all for faults within the tested clock period range on this unit. However, this unit shows two other interesting behaviors. First, no faults are triggered at less than 50ps of stress. This implies that the longest paths in the design are not true critical paths: they cannot be logically activated, or at least cannot be activated with these test programs. The slowest logic paths are used to check for valid virtual memory addresses for calculated load addresses (architecturally, this logic checks the so-called "VA-hole" in SPARC v9 [113]). The path lengths in the baseline case for this logic span to over 75ps before another exercised path is uncovered. Because the selected test programs purposefully generate valid virtual memory addresses, this logic is never activated. The next true longest paths are operand datapaths, starting at 50-75ps below the longest paths. These paths are exercised by the test programs.

Second, the execute ALU unit exhibits a stair step-like behavior for the architectural fingerprint coverage. This behavior is strongly correlated with the individual test programs. To analyze this further, Figure 42 shows the architectural fingerprint detection coverage for the execute ALU unit, separated by program. Most of the programs show the familiar behavior of sharply increasing coverage after a specific stress level. However, as shown in the figure, two programs, `mt_Ifill_L2` and `mtblkldst_loop`, are unable to propagate any true fault activations to architectural state, regardless of the evaluated stress levels.

The stair step behavior for the execute ALU unit is not, however, a deficiency in architectural fingerprint coverage. Only if the coverage were zero over all test programs should this be a concern. For error detection to succeed, only *one* program is necessary to activate and propagate each fault site. Furthermore, this situation can be rectified by writing more effective test programs that trigger the critical paths to architectural state when coverage is low.

The fetch control logic shown in Figure 41(c) differs from the prior two units because it is a pure control path. In this unit, the microarchitectural and architectural fingerprints closely follow each other in fault coverage. Because errors in the fetch stage are likely to directly affect architectural state. Furthermore, beginning at 50ps below the longest paths, timeouts are observed as well. This indicates the occurrence of deadlocks in the processor, which should be unsurprising because the fetch control logic is critical to fetching and executing instructions.

The fetch datapath unit shown in Figure 41(d) behaves similarly to the exu_alu datapath logic. Microarchitectural and architectural fingerprints show nearly equal coverage. However, because every program necessarily utilizes PC+4 and PC+branch offset logic, there is no stair step behavior evident in this unit. Furthermore, the processor continues fetching instructions—at invalid addresses—without immediately incurring deadlocks. Only when the processor is stressed hard enough (past 150ps) does the fetch logic fail badly enough to cause timeouts.

Finally, the thread switch logic shown in Figure 41(e) shows identical behavior for all three of the detection methods. This behavior results because, in practice, the thread switch logic is extremely sensitive to timing faults. The switch logic consists of multiple sparsely encoded state machines holding thread state (runnable, stalled, disabled, etc.) which can be easily corrupted by any propagated error. Unknown states are not handled in the RTL; therefore, the processor simply ceases scheduling the thread when it reaches an invalid state. Hence, the processor reaches a deadlock scenario almost instantly. This situation is easily detected by both types of fingerprints and by a simple timeout. This result differs from the prior section because new critical paths are formed with wearout, thus affecting execution even at lower levels of stress.

In summary, architectural fingerprints maintain coverage near that of microarchitectural fingerprints for at least one test program in all studied units. Furthermore, architectural fingerprints converge to the same coverage as microarchitectural fingerprints, making them as powerful as microarchitectural fingerprints for widespread wearout detection.

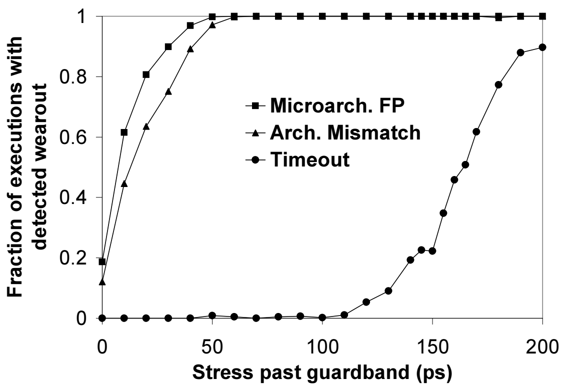
Detecting Silent Data Corruption

Next, this thesis analyzes the wearout detection results. The previous section shows that errors can be detected by timeouts alone in most units, if the processor is stressed hard enough. An interesting question is how far along will the processor continue retiring values architecturally under stressed conditions—potentially with silent data corruption. This separates out the situations where architectural and microarchitectural fingerprints are needed from where a simpler mechanism, such as waiting for a timeout, suffices.

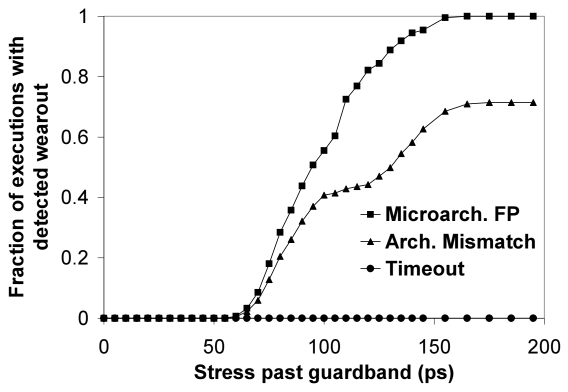
Figure 43 shows the contribution of architectural fingerprints, marked with triangles, when silent data corruption has been detected through a direct architectural fingerprint mismatch first, but excludes other cases. The excluded cases are when processor times out in execution (e.g., deadlocks on at least one thread) or the processor executes more instructions than expected (e.g., enters an infinite loop). This separates the sole contribution of silent data corruption detected by architectural fingerprints. The microarchitectural fingerprint, unchanged from the prior figure, again serves as perfect detection, while timeouts which are also unchanged, show a simpler detection method.

The results are divided into two distinct camps. First, the execute bypass, execute ALU and fetch datapath results in Figure 43(a)/(b)/(d) show no change in the coverage of architectural fingerprints. This result is expected because errors in these units are highly likely to first result in retiring incorrect data values without initially affecting the control flow of the program—and consequently, the number of instructions retired.

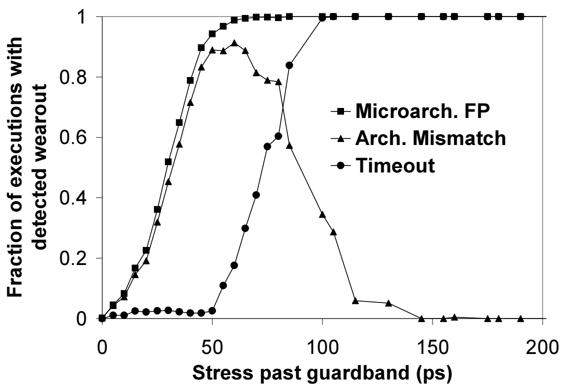
By contrast, the fetch control logic and thread select logic in Figure 43(c)/(e) both show a steep decrease in the coverage of architectural fingerprints when looking solely at silent data corruption. For the fetch control logic, this occurs in tandem with the rise in detected timeouts. The rise of architectural fingerprint detection indicates that the processor first shows moderate levels of silent data corruption in one test program, but as the stress level increases, the processor instead succumbs



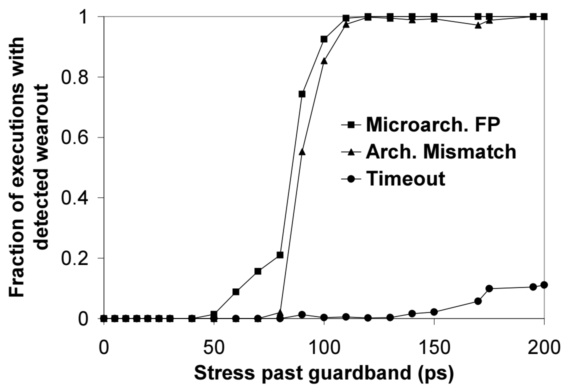
(a) byp



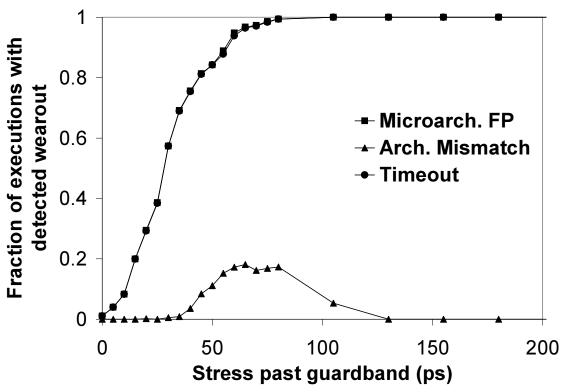
(b) exu_alu



(c) fcl



(d) fdp



(e) swl

Figure 43: Coverage of wearout faults where the contribution of silent data corruption detection with architectural fingerprints is isolated. Microarchitectural fingerprints and timeout results are unchanged.

to timeouts. For the thread switch logic, because the processor is so sensitive to changes in this logic, timeouts dominate. The small fraction of programs that show silent data corruption occur only because timing faults affect the stall signal paths, which temporarily re-enables the thread. However, the affected programs universally end in a timeout.

In summary, these results show that the strong detection results for architectural fingerprints are only partly due to directly detecting silent data corruption. The contribution of architectural fingerprint mismatches due to “collateral damage”, where the pipeline enters a timeout, is an important source of mismatches at the end of execution. While these effects can be detected by a simple mechanism such as an instruction timeout, detecting silent data corruption provides the earlier observation of wearout-induced timing faults at lower levels of stress that is needed to match high observability mechanisms such as microarchitectural fingerprints.

Detection with Architectural Test Programs

Because of the promising device wearout detection results with architectural fingerprints for wearout detection, this thesis now explores whether the same detection can be performed by simply looking at architectural outputs of a program. This mode of operation has the advantage that, if the architectural state can be easily exposed for comparison in software, this technique can be used on existing processors without additional hardware (e.g., fingerprint compaction). While simpler, the main disadvantage of software-based architectural error detection is additional masking by the program. Because there are intermediate results in execution—values later overwritten in registers and memory—the final architectural state is subject to program-level masking. Therefore, the coverage of software-based architectural error detection should be strictly less than or equal to that of architectural fingerprints, differing by the degree of program-level masking.

A second disadvantage of the software-based architectural error detection is that in order to perform the comparison, the architectural registers and memory must remain accessible by software. However, the nature of the FIRST procedure can make this impractical. If the core has deadlocked after attempting to run a test program in marginal conditions, the core may require a reset before registers and memory can be read for comparison. Unfortunately, reset procedures typically cause the processor to run initialization code that overwrites portions of architectural registers and memory. Therefore, this low-cost error detection mechanism may be impractical. However, to determine error detection capabilities of this mechanism, this evaluation optimistically assumes that the complete, final architectural register and memory state remains accessible following the test.

For this evaluation, the final architectural register and memory state of the program (either by timeout or by completing execution) is assumed accessible by a software comparison mechanism that compares the processor's architectural state to error-free reference outputs. A C program accumulates the register updates produced by the OpenSPARC processor under the range of stress levels into architectural registers and a memory image and compares that image with the same program's architectural output during error-free operating conditions.

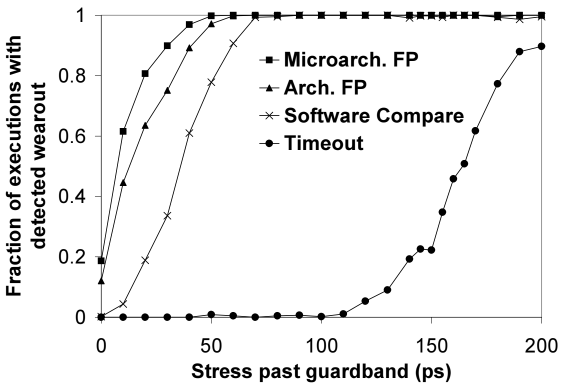
Figure 44 shows the coverage of device wearout over the seven workloads and instances of wearout using the three detection mechanisms presented earlier and the additional proposed software-based architectural error detection mechanism. As expected, the coverage is consistently less than or equal to architectural fingerprint coverage for a given stress level. For the fetch control logic, fetch datapath, and thread switch logic, the detection capabilities are almost identical to architectural fingerprints. This outcome is unsurprising because these units produce serious control flow errors in execution when subjected to faults. By contrast, the bypass and execute units continue to execute the program with SDC when subjected to faults. In these units, the final architectural results are more clearly masked by the test programs. For example, only two test programs manage to avoid partially program-level masking with the execution unit. This difference can be largely eliminated by re-writing test programs that program level masking (e.g., by preserving intermediate values in memory) at the cost of longer execution time and a larger test output for comparison at the end of execution.

In summary, this result shows that software-based architectural error detection is almost as effective as architectural fingerprints for early widespread device wearout detection. The results can be further improved with test programs that avoid masking.

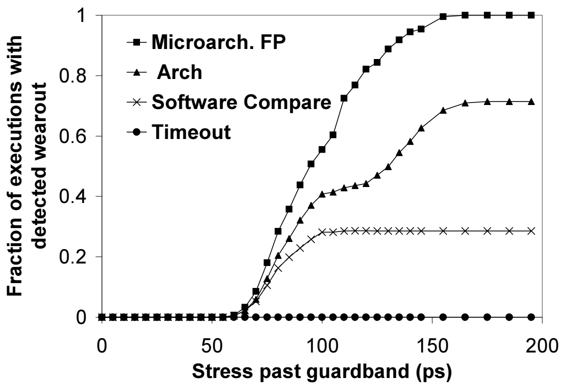
7.5.3 The Persistent Nature of Wearout Faults

While radiation-induced soft errors exhibit high degrees of masking on architectural, microarchitectural, and logical levels in RTL simulation, wearout fault models show high coverage with architectural detection mechanisms. This section quantitatively discusses the differences between the two fault models and explains the disparity in architecture-level fault coverage.

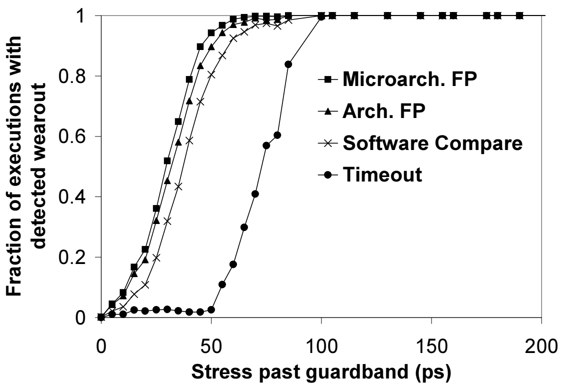
The wearout faults have two key properties that make them easier to detect architecturally than radiation-induced soft errors: higher spatial distribution of faults and higher temporal repetition



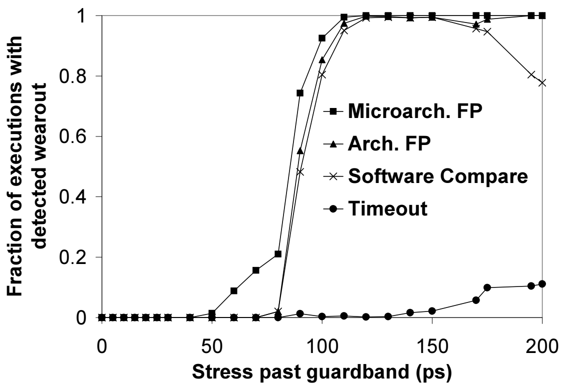
(a) byp



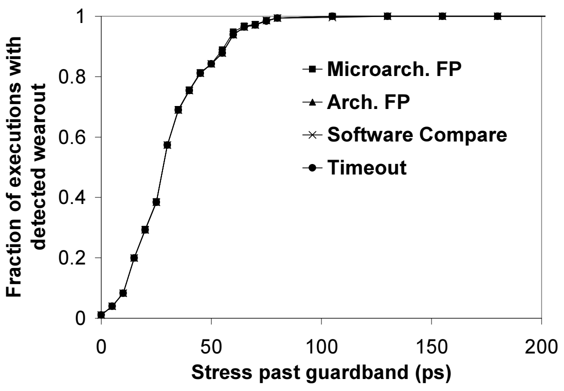
(b) exu_alu



(c) fcl



(d) fdp



(e) swl

Figure 44: Coverage of wearout faults with software-based architectural error detection. Microarchitectural and architectural fingerprints and timeout results are unchanged.

of the same underlying fault. The wearout faults are best described as *intermittent* faults in the literature [12]. The intermittent faults repeat, given the correct environmental conditions and input stimulus. This behavior is in contrast to radiation-induced soft errors that strike once and affect a

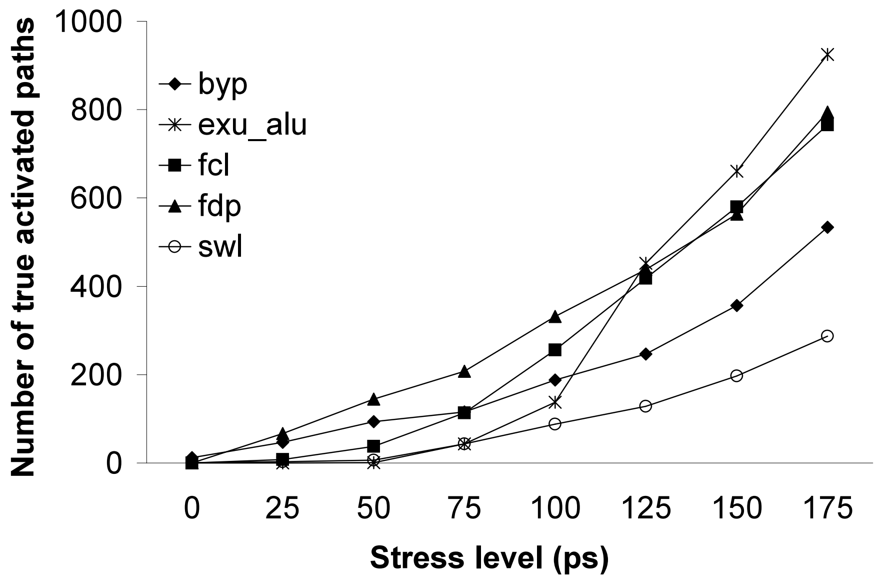


Figure 45. The number of true activated paths as a function of stress level for each unit.

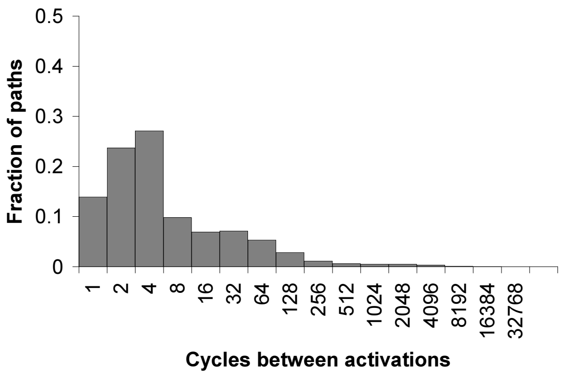
small number of state bits.

Spatial distribution. Because modern designs typically have several critical paths that are similar in length and levels of wearout are thought to be roughly uniform across a unit, multiple fault sites can appear simultaneously when reducing the guardband. This increases the chance that at least one erroneous path remains unmasked.

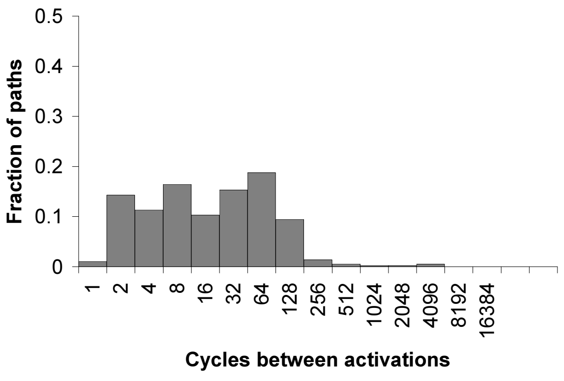
Figure 45 quantifies the average number of unique paths that were truly activated for each unit as a function of stress level. As the stress level increases, all units show a clear trend of increasing numbers of paths that fail to meet timing. This number quickly grows to hundreds of paths with small levels of stress, which greatly increases the chances of an error being made visible architecturally.

Temporal Repetition. Wearout faults are also activated when the environmental and logical conditions are fulfilled—situations that can occur repeatedly during execution. This repetition provides further opportunities for the errors to avoid further masking on logical and microarchitectural levels.

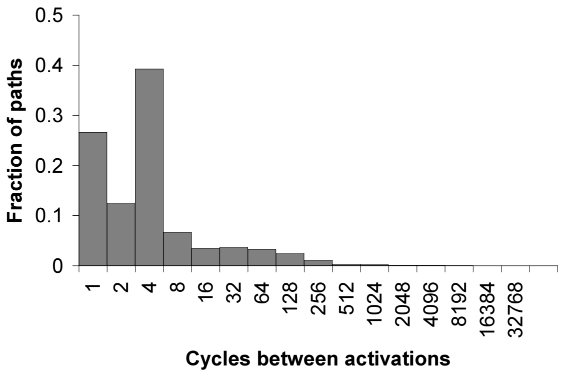
Figure 46 shows, for all true activated errors in each unit, the distribution of time between successive activations for a stress level of 200ps. This measure indicates the frequency at which faults are reactivated. In stark contrast to soft errors, where radiation-induced errors are typically



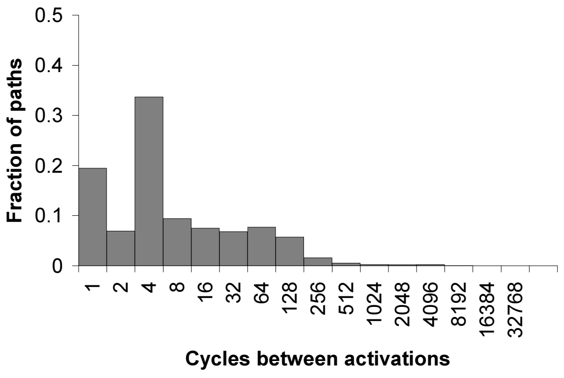
(a) byp



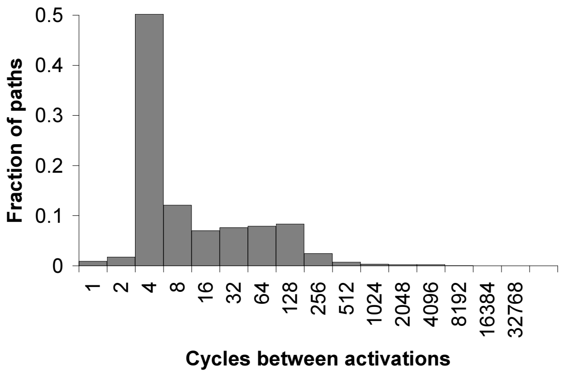
(b) exu_alu



(c) fcl



(d) fdp



(e) swl

Figure 46: Histograms showing the distribution of activations (in cycles) between successive true activations of the same path over all test programs, separated by unit.

single-event upsets, a majority of wearout errors reoccur within 16 cycles. This provides repeated opportunities for an error that may have been masked to appear as an architectural error.

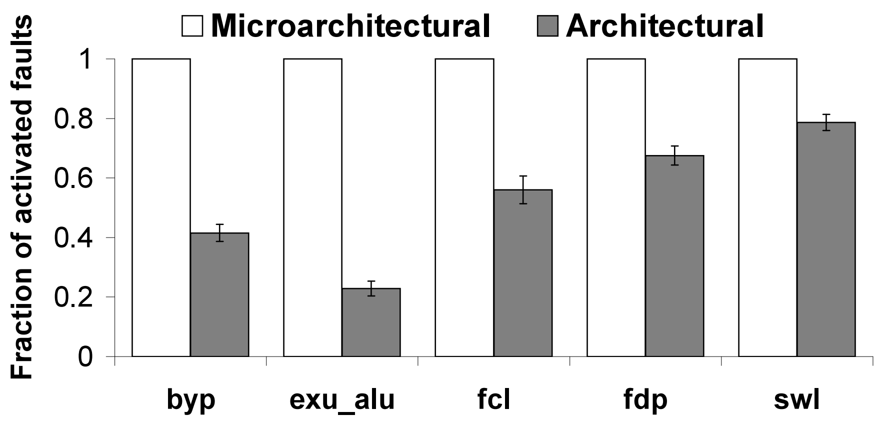


Figure 47: Detection coverage for a single wearout error site in each unit for microarchitectural and architectural fingerprints.

7.5.4 Isolated Wearout Faults

Section 7.2 outlines an argument for expecting multiple, widespread wearout faults to grow concurrently within a processor. However, it is fair to ask what coverage is possible if only one device initially experiences wearout. This study briefly characterizes the detection coverage for a single wearout fault.

Figure 47 shows the detection coverage for microarchitectural and architectural fingerprints monitoring a single wearout fault. In each simulation (over the same set of tests as in the prior section), a single path is chosen to be the most critical path in the unit, uniformly selected over all paths in the unit. The stress conditions are set to trigger only that path; however, the fault may be activated repeatedly during execution. The figure only reports runs where an error is truly activated by the test programs, therefore microarchitectural fingerprints are guaranteed to have observed the error. This filtering procedure eliminates the quality of the test program as a factor in the result. When a single error exists, the test program must exercise the correct transition without logical masking to activate the fault. However, these test programs are not designed to activate specific fault sites, therefore a majority of the executions in this experiment fail to produce errors.

The results in Figure 47 show coverage of the single wearout error for microarchitectural and architectural fingerprints. Each bar indicates the fraction of errors that are detected by each detection mechanism. The microarchitectural fingerprints are guaranteed to observe the errors, except in the unlikely event of aliasing. By contrast, the architectural fingerprints observe the same errors only 48% of the time, on average. This indicates that architectural fingerprints are less powerful than microarchitectural fingerprints when only one fault is present and activated. However, this should not be taken as a blanket conclusion that architectural fingerprints are ineffective in this scenario. Instead, higher-quality test programs that are specifically written to propagate a wide range of errors to architectural state can improve the coverage of these errors, as is commonly done for functional tests in manufacturing test.

7.6 Conclusion

This thesis proposed FIRST, an early wearout detection procedure. FIRST avoids the complex or area-inefficient detection mechanisms associated with just-in-time error detection by running functional tests and exposing faults by changing the operating environment (e.g., voltage, frequency, temperature) of the processor to expose marginal circuits that are affected by the early stages of wearout. Through an RTL fault injection-based evaluation over a selection of representative units, this thesis shows that microarchitectural fingerprints are effective at detecting errors from both isolated and widespread wearout. Furthermore, architectural fingerprints are as effective as microarchitectural fingerprints for detecting errors from widespread wearout.

Chapter 8

Related Work

This chapter places the work from this thesis in the context of related work and products in the areas of reliable computer system design using redundant execution for concurrent error detection and wearout detection. The fingerprint concept studied in this thesis provides lightweight error detection for errors affecting the processor pipeline through efficient compaction and comparison of carefully constructed hashes that represent the architectural or microarchitectural state updates produced by a processor over an interval of execution. Fingerprints provide a timely and bandwidth-efficient mechanism for comparing large amounts of state, as is necessary for making error-free checkpoints, and checking or testing the correct operation of a processor.

This chapter is organized by the applications of fingerprints. The Reunion execution model in Chapter 5 proposes a complexity-effective set of changes to existing non-redundant multiprocessor designs that allow concurrent error detection and recovery using architectural fingerprints for lightweight error detection. Recent developments in concurrent error detection and reliable system design are discussed. The FIRST procedure in Chapter 7 investigates the use of architectural and microarchitectural fingerprints for early device wearout detection by tracking the development of timing faults by periodically testing processor cores with functional tests in marginal operating conditions. The FIRST procedure is compare to recent proposals for competing and complementary wearout detection techniques.

8.1 Concurrent Error Detection

Concurrent error detection with lockstep. Traditional approaches to concurrent error detection originate from mainframe designs. Recent IBM zSeries mainframe designs detect datapath errors using custom dual-lockstepped pipelines whose results are directly compared on every instruction [101]. This configuration reaches the ideal limits of an architectural fingerprint—minimal error detection latency, ample on-chip bandwidth, and no possibility of aliasing because values are directly compared.

The Tandem NonStop approach uses lockstepped commercial processors with chip-external comparison on the memory bus to detect errors and initiate rollback-recovery [16]. Because of the unbounded detection latency associated with chip-external detection, the NonStop systems rely upon software checkpoints of the system state, which requires a custom operating system and applications.

Instead of requiring custom software for rollback-recovery, the lockstepped Stratus ftServer supports commodity operating systems on commodity processors by performing forward error recovery [112]. With this recovery the system can avoid checkpoints and thus tolerates long detection latencies; however, the system's integrity is critically dependent upon the existing error detection hardware to identify correctly the failing unit. Error detection and isolation in the ftServer is performed through a combination internal processor error signals (e.g., parity detection) and output comparison at I/O requests. The system pauses execution on error detection and continues execution with the remaining units, following isolation. In the absence of an error signal or clear diagnostic evidence of the failing unit, which is likely with soft errors, the correct execution can only be determined through voting with triple modular redundancy. Such a solution incurs at least 50% higher system hardware costs than dual-modular redundant systems.

Maintaining lockstep in future systems, even on a single chip, requires fighting increasingly burdensome technology trends, with concerns over wire delay, process variations, and defects making lockstep difficult to maintain [66]. Furthermore, lockstep requires deterministic execution, which has increasing validation expenses as systems evolve. While manufacturing test typically drives validation for determinism today, the needs of manufacturing test are limited to a relatively short runtime and a predetermined set of applications under carefully controlled conditions. System-wide validation of asynchronous interfaces, shared components (e.g., buses and interconnects), error handling and correction routines, and thermal controls make lockstep increasingly difficult to maintain [18].

Redundant Multithreading. In an effort to relax the lockstep requirements, industry and researchers have investigated a wide variety of redundant multithreaded (RMT) execution. In these systems, two or more threads execute the same program redundantly in time or space, but do not require microarchitectural determinism, as is needed for lockstep. Instead, the redundant threads are required to produce the same architectural results, but not necessarily in the same way. The key differences in these proposals lie with 1) coordinating replicated inputs across the redundant executions and 2) comparing the outputs with error detection mechanisms. Architectural fingerprints are appropriate for RMT approaches because they only require architectural determinism—that redundant executions produce the same results—but specifically allow those results to be produced through different methods.

Within industry, the Tandem NonStop Advanced Architecture (NSAA) has moved from lockstepped redundant execution to RMT across loosely coupled processor pairs [18]. NSAA replicates processors and memory across physically separate units and compares execution through I/O transactions at a custom I/O controller. As with prior NonStop systems, a custom operating system coordinates delivery of external interrupts to the processors and recovery occurs through software checkpoints. To keep the redundant executions synchronized at the I/O level, the processors' speculation support is disabled to prevent the processors from producing different page faults (which would appear as erroneous outputs at the I/O controller). Finally, this system does not support shared memory.

Marathon everRun servers implement RMT with commodity operating systems by using virtual machine monitors to hide the input replication and output comparison [63]. The virtual machine coordinates external interrupt delivery by single-stepping the processors to predetermined time quanta and delivering the interrupts only at these points. Errors are detected at I/O requests, which are trapped and compared by the virtual machine. As with the lockstepped Stratus machines, diagnostics are used to determine the failing unit and execution halts, or if possible, continues with working units.

The Marathon, Stratus and the NonStop systems have the common advantage that they can use existing commodity microprocessor designs. However, this results in the common drawback that none can conclusively determine, at a given point in execution, whether undetected errors exist in the core's architectural output because of the unbounded error detection latency. Thus, they depend on software recovery mechanisms or forward error recovery that are tolerant of long error detection latencies, but require custom operating systems or high hardware costs to protect against soft errors in the pipeline. By contrast, architectural fingerprints provide a light-weight architectural mechanism to efficiently and quickly detect pipeline errors at a fixed point in execution, as is needed for producing error-free checkpoints. This enables RMT with fine-grained hardware-based checkpoint recovery and OS-transparent redundant execution without lockstep.

Research proposals investigate RMT in two main contexts. First, redundant threading within a single core is studied heavily in recent research [36, 72, 81, 85, 88, 90, 106, 115]. These systems reuse existing microarchitectural throughput mechanisms, such as multithreading and speculation to provide redundant execution and recovery in a shared pipeline. The load inputs are typically replicated through a load value queue (LVQ) which records a sequence of loads and replays them for a consuming thread [88] and outputs are compared through direct comparison of stores (if just detection is desired) or architectural register values and stores (if recovery is also desired). Two main drawbacks of these techniques are the microarchitectural implementation complexity and its resulting validation issues—these techniques and the LVQ all require significant changes to an already-complicated out-of-order processor core—and the 20-30% performance penalty of redundant execution on a shared pipeline [106].

Second, researchers investigate redundant execution across cores in a chip multiprocessor (CMP) [40, 56, 72, 114], including the Reunion proposal in Chapter 5. As compared to within-core RMT, these proposals have the added burden of replicating inputs and comparing outputs across physically-distributed cores. Both of these tasks require significant cross-core bandwidth, matching the bandwidth of the L1 cache ports and architectural register file to copy inputs from one core and replay them for the other, while comparing the architectural outputs. However, existing datapaths do not support this level of cross-core bandwidth, so expensive, wide buses must be added and each core must be modified to support the strict input replication (LVQ) and direct output comparison [40, 72]. Furthermore, none of these proposals address redundant execution in shared memory, which is now a necessary feature for mainstream computer systems.

The Reunion proposal from this thesis addresses the input replication, output comparison, and shared memory problems. Reunion is predicated on the observation that in the common case, even for shared memory programs, two redundant executions will receive the same load values. In the uncommon case, soft error detection already observes this situation and rollback-recovery can resolve the difference. This leads to a complexity-effective design that provides redundant execution without changing the complicated parts of the microarchitecture. Furthermore, with architectural fingerprints, the error detection latency is timely enough to permit fine-grained recovery with existing speculation mechanisms, while the bandwidth overhead can be satisfied with existing on-chip interconnects. The design presents a single logical processor to the system, which means that software changes are not necessary to permit redundant execution. Finally, because the microarchitecture requires small changes to portions of the pipeline and cache off the critical path, the design also allows a dual-use redundant and non-redundant design with little incremental hardware cost.

Comparison Bandwidth Reduction. Researchers have looked at reducing the comparison bandwidth. For detection-only operation, researchers have proposed directly comparing store addresses and values [72, 88]. Alternatively, for recovery, the death and dependence-based checking elision (DDBCE) compression in [40] follows errors that propagate through architectural data dependencies. Only results terminating a dependence chain must be compared. However, chains can be ended early by instructions that potentially mask an error; therefore, the authors are only able to elide about 20% of instruction comparisons in SPEC CPU 2000. Architectural fingerprints easily reduce comparison bandwidth by orders of magnitude through compaction within and across instruction results. This compaction enables comparison across cores within and across chips.

Self-checking and hardened logic. An alternative to redundant execution for soft error protection is to build structures that are more resilient. As early as the 1950's, architects have implemented various forms of self-checking such as parity prediction and residue codes in arithmetic units and processor logic of mainframe systems [100]. These techniques provide protection for specific logic units within the pipeline. However, such designs prove too expensive for commodity processors because of their impact on clock frequency and area overhead.

Circuit-level protections, such as soft error-tolerant latches, also provide protection across the pipeline with minimal timing overhead by time-shifting combinational logic outputs across two latches [70]. However, this design provides protection for glitches that propagate to latches in the pipeline and can more than double the latch area overhead.

By contrast, architectural fingerprints, which inspect state at the retirement stages of the pipeline, provide comprehensive detection of architectural errors that occur within the pipeline without unit-specific protection. The cost of this protection, however, is that redundant execution support is necessary.

Signature-based error detection. Researchers have looked at using signatures of control flow for microprocessor error detection [119]. This technique depends upon compiler-inserted data to compute signatures of legal control flow graphs, along with a watchdog processor that does an on-line comparison of actual control flow to that specified in the signature. Unfortunately, this work cannot easily be extended to data values, because compilers can only determine allowable data values in limited situations. Architectural fingerprints, by contrast, are generated by hardware at runtime, and are compared with a redundant unit and can therefore detect errors in both control and data flow. In both cases, the signature-based detection mechanisms require determinism on the architectural level to avoid false mismatches.

Architectural Fingerprints. The TRUSS project at Carnegie Mellon University uses architectural fingerprints to compare redundant processors in a distributed shared memory machine [39]. Recently, other researchers have also applied architectural fingerprints to allow aggressive overclocking [41] and to allow redundant core pairs to be decoupled and separated across cores in Reunion-based process in a CMP [56]. These techniques allow simultaneous improvements in reliability and single-thread performance over non-redundant systems.

8.2 Wearout Detection

This section discusses the related work in early wearout detection and processor manufacturing test. Existing design-for-test (DFT) hardware addresses a problem similar to that faced by fingerprints, namely efficient compaction of large amounts of state to allow efficient and fast detection of errors within that state. The wearout error detection attempts to find evidence of faults in a processor core, as done in traditional manufacturing test, except the test must occur at runtime in a customer environment instead of in a high-volume tester.

Scan and scanout. Modern microprocessors include full-hold scan and partial chip scanout technology [55]. The scanout hardware forms the basis of the microarchitectural fingerprints discussed in Chapter sec:mf. Full-hold scan chains link all pipeline latches in a chain that can then create a snapshot of the global latch state, which is then scanned out. The scan operation provides high observability of internal state and is used in manufacturing test and debug, however it is destructive (the core clock is paused, original latch values are overwritten and execution cannot continue), requires expensive testers to control the test, and has high bandwidth and storage overheads. Scanout chains are added to existing logic, providing a non-destructive and compact summary of execution over time [24, 55]. Scanout can operate without a tester, which enables runtime applications. However, even with highly controlled environments and automated testers, non-determinism in the circuit under test can cause both scan and scanout to falsely declare an error [80].

Researchers have also looked at scanout-like devices for on-line error detection [107]. Unlike microarchitectural fingerprints, this technique compares a constant stream of monitored nodes to a reference model. Hence, this technique cannot tolerate the non-determinism that plagues scanout.

Output compaction and non-determinism. Researchers have investigated output compaction and non-determinism tolerance techniques in the context of manufacturing test. With these techniques, the system under test runs pre-determined test vectors or functional tests in an automated tester. Test output is typically compacted with parity tree-like structures [86], LFSRs, and MISRs [13, 52]. By determining a priori when non-determinism will be exhibited in output sequences, non-deterministic signals can be masked out [80] or corrected with techniques derived from coding theory [69] if the test output is known and the number of non-deterministic signals is small. Mitra and Sup Kim's X-compact technique is applied in this thesis as a strong spatial compactor for architectural fingerprints, without taking advantage of its correction properties.

Compaction researchers also study designs that guarantee zero aliasing in the signature output. Chakrabarty et al. investigate spatial output compaction with zero aliasing [26]. A key limitation to this work, however, is that the error space must be known prior to the test in order to achieve the guarantee, a task that becomes difficult when large numbers of errors are expected concurrently.

The key difference between these compaction techniques and microarchitectural fingerprints is using the designer's knowledge of the circuit to mask values that are non-deterministic before they reach the compaction stage. The output masking and correction techniques focus on blocking non-deterministic inputs at the input to the compactor with special masks for each test. If non-deterministic sites are identified beforehand, microarchitectural fingerprints can eliminate the non-determinism without needing customization for each test.

Wearout detection. Recently there has been much activity in the area of predicting and detecting the onset of wearout. Srinivasan et al. [111] propose an empirically derived environmental model for predicting circuit failure and enabling spare units. Empirical models provide a statistical prediction of wearout's onset; however, they do not directly observe wearout on the core and therefore may be overly conservative. Techniques such as sparing can improve the useful operating lifetime of a processor by replacing failing units with spare, undamaged units.

Techniques such as Razor [33] and circuit failure prediction [2] allow circuits to monitor themselves constantly for timing violations at runtime using special double-sampling latches. The Razor technique allows dynamic voltage scaling to reduce power until the circuit begins to produce incorrect results. Circuit failure prediction aims to extend processor lifetimes by gradually increasing the frequency guardband when device wearout begins to occur, thus absorbing the slowdown and preserving correct operation.

Blome et al. [19] inserts delay detection units throughout the pipeline to sample, measure, and track the progression of delay from device wearout. In particular, this technique presents a simple hardware monitor that accumulates and averages delays across the chip to identify gradual slowdown. Similar work has been proposed with canary circuits that are designed to fail before other devices on the chip [87], however these techniques rely on the canary circuit to be stressed as heavily as the rest of the chip. Furthermore, the canary circuit must experience the same degree of wearout as the most vulnerable devices in the design.

Built-in self-test (BIST) techniques implement test hardware directly in the design, avoiding the need to propagate errors to external pins and reducing the dependence on external testers [1]. These structures constitute a fixed hardware overhead that is typically only used during manufacturing test and is otherwise disabled at runtime. Therefore, techniques that can reuse these structures at runtime are extremely attractive.

Shyam, et al. [98] recently proposed a continuous online BIST to detect failures in the field using distributed test vectors that are pushed through major microarchitectural components on idle cycles. Their technique requires custom BIST hardware constructions and test vector ROMs to be designed for each piece of the design. While microarchitectural fingerprinting requires designer knowledge for each unit, it also has a common architecture that can be used throughout the design. This allows a single common set of rules for design engineers to follow.

Li et al [59] propose using existing scan chains and externally-stored tests to periodically run thorough structural and functional tests on the core microarchitecture at runtime in a procedure called CASP (Concurrent Autonomous chip self-test using Stored test Patterns). The procedure is initiated similar to FIRST by periodically taking cores off-line and running self-test programs and structural test patterns through scan chains, using an added test controller. Because testing runtime is more relaxed than in high-volume manufacturing test, the selected tests can have high coverage that meets or exceeds that of traditional manufacturing test. As proposed, CASP does not reduce the frequency or voltage guardbands, as is done in FIRST, however doing this could improve timely detection of wearout before it affects execution at normal operating frequencies and voltages.

The FIRST procedure presented in this thesis provides an alternative and potentially complementary method for identifying and predicting wearout. Architectural and microarchitectural fingerprints provide a compact method for storing the outcome of a test, which avoids program-level masking and speeds the execution of functional tests. FIRST helps identify developing faults earlier and more precisely by proactively removing the guardband (as opposed to waiting for runtime latch timing failures) and looking broadly across the design for actual microarchitectural and architectural errors (as opposed to sampling a small number of signals). Furthermore, analysis and storage of test results from FIRST can be stored and processed in software, which simplifies long-term tracking of device wearout.

When comparison is performed with architectural fingerprints or architectural state comparison, FIRST is limited to running functional test programs. This limitation has the drawbacks associated with functional test fault coverage, which is traditionally lower than structural tests in the manufacturing test regime. However, this concern may be unjustified if, instead of emerging as isolated path delays, wearout appears a widespread slowdown simultaneously affecting a multiple paths at once. In this situation, as shown in Chapter 7, even a small number of short functional verification tests are effective at revealing widespread device wearout architecturally.

Chapter 9

Conclusion

As CMOS technology continues scaling to smaller dimensions, aggressive processor designs are widely expected to confront increasing rates of both soft errors and device wearout. Without techniques to detect and recover from these errors, microprocessors will become increasingly unreliable. This thesis develops the concept of fingerprints to detect soft errors and device wearout in the processor pipeline. Architectural fingerprints are signatures of in-order architectural state updates collected during execution by a small unit added at the retirement stages of the pipeline. These fingerprints provide lightweight detection of architectural errors over a chosen interval of execution by a simple comparison. Microarchitectural fingerprints utilize existing design-for-test hardware to collect an in-depth summary of internal microarchitectural state updates. These signatures provide a higher level of coverage for internal nodes as compared to architectural fingerprints; however, with more stringent requirements for determinism.

This thesis investigates the design and implementation of architectural fingerprints in detail. The key results of this study show that architectural fingerprints provide effective lightweight detection architectural errors through soft error injection in a commercial processor RTL model. Furthermore, synthesis results demonstrate that hardware for architectural fingerprints requires less than 4% of the logic area of an already-simple commercial scalar pipeline. A hash design study shows that the combination of an X-compact-like spatial compactor and a MISR temporal compactor yield an area and timing-efficient implementation with aliasing properties approaching those of an ideal cyclic redundancy check.

This thesis also shows two applications for fingerprints that help preserve reliability in microprocessors. The Reunion execution model provides a formal framework for complexity-effective redundant execution in shared memory systems to provide soft error tolerance. This study shows that the same mechanism needed for soft error detection—architectural fingerprints—can be applied to detect input incoherence during redundant execution. Furthermore, with simple changes to the operation of the cache controllers, the execution model permits correct redundant execution in designs with shared memory, while avoiding changes to the complicated system components such as the out-of-order execution, cache coherence protocol, and memory consistency model. A cycle-accurate simulation-based evaluation of a chip multiprocessor shows that Reunion incurs a modest 5-6% performance overhead over more complicated concurrent error detection designs with strict input replication.

Finally, this thesis proposes and evaluates the FIRST procedure to detect early signs of device wearout. FIRST places the processor in marginal operating conditions to identify changes, over time, in the speed of devices in the pipeline. Fingerprints provide a fast, bandwidth-efficient mechanism for comparing the results of these repeated tests at runtime. The results show that microarchitectural fingerprints are effective at observing both single and widespread wearout fault development, while architectural fingerprints are as effective as microarchitectural fingerprints for detecting widespread wearout fault development.

9.1 Future Work

The work presented in this thesis opens new questions that are ripe for investigation by future researchers.

Reunion. The Reunion execution model has been thoroughly investigated for parameters on today's and near-future chip multiprocessors with short latencies between cores and precise-exception rollback. However, the design tradeoffs can change in other system architectures, such as distributed shared memory systems. In such systems, the latency between processors is orders of magnitude higher, while the bandwidth is lower, therefore fingerprint comparison intervals must grow commensurately. Furthermore, when processors include checkpoint capabilities they can potentially execute more instructions between comparisons, which opens up the possibility for additional data races and the need for efficient rollback-recovery protocols. There is evidence that indiscriminate application of the unoptimized Reunion recovery protocol seriously impacts performance [56], however appropriate changes to checking and recovery should be able to eliminate this loss.

Reunion is primarily motivated by the need for reliable execution. However, the current implementation will roughly require double the processor power budget and yield less than half the performance of two individual cores. Paceline [41] investigates over-clocking redundant cores in a slipstream-like configuration to improve performance. Reunion can be applied to do similar aggressive execution at different voltage and frequency corners. Halving the dynamic power at constant performance, to match non-redundant power, is an unrealistic goal, but the overhead need not be twice the non-redundant power.

Correlated failures in redundant cores with aggressive execution are also a concern. However, the cores need not be identical for several reasons. The increasing trend of within-die variability may be a blessing in disguise, heterogeneous chip multiprocessors can naturally avoid some correlated failures, and sparing mechanisms for lifetime reliability may provide enough diversity to avoid correlated failures. Furthermore, running one core with more-conservative margins than the other will reduce power saving, but can also change the intermittent failures enough to be consistently detectable. If these failures are reproducible, predictors also may be used to indicate when more conservative execution is needed. The correlated failures warrant further studies to characterize how processors actually fail—gracefully degrade or fail abruptly—in marginal conditions.

FIRST. FIRST has a number of avenues for future work. The general connection between wearout onset and changing architectural fingerprints has been made in this thesis. However, the model can be strengthened to yield actual predictions about the extent of wearout, time left before hard breakdown, both inferred from changes in the fingerprint with time and environmental settings.

In addition to predictions about failure time, the information gained from FIRST can potentially extend the lifetime of failing processor components. The growth of many wearout faults has been shown to be dependent on environmental conditions (e.g., voltage and temperature). Therefore, when developing wearout faults are detected, the processor's voltage and frequency can be dropped to more-conservative set points. In this way, performance can be traded for longer processor lifetime from both the combined benefits of both an increased guardband and slower fault growth rates.

FIRST with microarchitectural fingerprints depends heavily on the microarchitecture remaining a constant over time (except for wearout). However, lifetime and defect-aware processors will change over time as spare units are swapped in for failing units. Methods need to be developed for tolerating a microarchitecture that reconfigures with time due to repair. By contrast, for architectural fingerprints, the FIRST procedure should be cognizant of microarchitectures that contain spare units and ensure that errors in redundant and spare units can be detected. The impact on FIRST of a microarchitecture that changes over time should be investigated.

More information on wearout of non-critical paths can be gleaned from FIRST than just the initial change in a fingerprint's signature at specific condition. The proposed FIRST procedure only stresses the processor core until the first mismatch. However, there may also be "plateaus" of path lengths where, at a stress level higher than the first signature mismatch, the fingerprint again becomes a stable value—different from the error-free value—yet still stable. Assuming variability does not destroy this opportunity; additional evidence of wearout can be ascertained for shorter paths that never slow down enough to become critical paths before entering hard breakdown.

References

[1] Miron Abramovici, Melvin A. Breuer, and Arthur D. Friedman. *Digital Systems Testing and Testable Design*. IEEE Press, revised printing edition, 1990.

[2] Mridul Agarwal, Bipul C. Paul, Ming Zhang, and Subhasish Mitra. Circuit failure prediction and its application to transistor aging. In *Proceedings of the 25th Annual IEEE VLSI Test Symposium (VTS-07)*, May 2007.

[3] Anastassia Ailamaki, David J. DeWitt, Mark D. Hill, and David A. Wood. DBMSs on a modern processor: Where does time go? In *The VLDB Journal*, pages 266–277, September 1999.

[4] Haitham Akkary, Ravi Rajwar, and Srikanth T. Srinivasan. Checkpoint processing and recovery: Towards scalable large instruction window processors. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 36)*, Dec 2003.

[5] Guido Albertengo and Riccardo Sisto. Parallel CRC generation. *IEEE Micro*, 10(5):63–71, Oct 1990.

[6] AMD Corp. *BIOS and Kernel Developer's Guide for AMD NPT Family 0Fh Processors*, revision 3.04 edition, Dec 2006.

[7] AMD Corp. *AMD Opteron Processor Product Data Sheet*, revision 3.23 edition, Mar 2007.

[8] Hisashige Ando, Ken Seki, Satoru Sakashita, Masatosh Aihara, Ryuji Kan, Kenji Imada, Masaru Itoh, Masamichi Nagai, Yoshiharu Tosaka, Keiji Takahisa, and Kichiji Hatanaka. Accelerated testing of a 90nm sparc64 v microprocessor for neutron ser. In *Proceedings of the 3rd IEEE Workshop on Silicon Errors in Logic – System Effects (SELSE-3)*, Apr 2007.

[9] Artisan Components. *TSMC 0.18μm Process 1.5-Volt (Low-Voltage) SAGE-X Standard Cell Library Databook*, 3.0 edition, Feb 2002.

[10] Todd M. Austin. DIVA: A reliable substrate for deep submicron microarchitecture design. In *Proceedings of the 32nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 32)*, November 1999.

[11] Alejandro Avellan and Wolfgang H. Krautschneider. Impact of soft and hard breakdown on analog and digital circuits. *IEEE Transactions on Device and Materials Reliability*, 4(4):676–680, Dec 2004.

[12] Algirdas Avižienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, Jan-Mar 2004.

[13] Carl Barnhard, Vanessa Brunkhorst, Frank Distler, Owen Farnsworth, Brion Keller, and Bernd Koenemann. OPMISR: the foundation for compressed ATPG vectors. In *Proceedings of the 2001 International Test Conference*, October 2001.

[14] Luiz Andre Barroso, Kourosh Gharachorloo, and Edouard Bugnion. Memory system characterization of commercial workloads. In *Proceedings of the 25th Annual International Symposium on Computer Architecture (ISCA)*, pages 3–14, June 1998.

[15] Luiz Andre Barroso, Kourosh Gharachorloo, Robert McNamara, Andreas Nowatzyk, Shaz Qadeer, Barton Sano, Scott Smith, Robert Stets, and Ben Verghese. Piranha: A scalable architecture base on single-chip multiprocessing. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, June 2000.

[16] Joel Bartlett, Jim Gray, and Robert Horst. Fault tolerance in tandem computer systems. Technical Report TR-86.2, HP Labs, 1986.

[17] Robert Baumann. Soft errors in advanced computer systems. *IEEE Design and Test of Computers*, pages 258–266, May-June 2005.

[18] David Bernick, Bill Bruckert, Paul Del Vigna, David Garcia, Robert Jardine, Jim Klecka, and Jim Smullen. Nonstop advanced architecture. In *Proc. Intl. Conf. Dependable Systems and Networks*, Jun 2005.

[19] Jason Blome, Shuguang Feng, Shantanu Gupta, and Scott Mahlke. Self-calibrating online wearout detection. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 40)*, Dec 2007.

[20] Darrell Boggs, Aravindh Baktha, Jason Hawkins, Deborah T. Marr, J. Alan Miller, Patrice Roussel, Ronak Singhal, Bret Toll, and K. S. Venkatraman. The microarchitecture of the intel pentium 4 processor on 90nm technology. *Intel Technology Journal*, 8(1), Feb 2004.

[21] Shekhar Borkar. Design challenges of technology scaling. *IEEE Micro*, 19(4):23–29, Jul-Aug 1999.

[22] Shekhar Borkar. Designing reliable systems from unreliable components: the challenges of transistor variability and degradation. *IEEE Micro*, 25(6):10–16, Nov-Dec 2005.

[23] Fred A. Bower, Daniel J. Sorin, and Sule Ozev. A mechanism for online diagnosis of hard faults in microprocessors. In *Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 38)*, Dec 2005.

[24] Adrian Carbine. Scan mechanism for monitoring the state of internal signals of a VLSI microprocessor chip. US Patent 5,253,255, Oct 1993.

[25] Jonathan R. Carter, Sule Ozev, and Daniel J. Sorin. Circuit-level modeling for concurrent testing of operational defects due to gate oxide breakdown. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE'05)*, 2005.

[26] Krishendu Chakabarty and John P. Hayes. Test response compaction using multiplexed parity trees. *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, 15(11), Nov 1996.

[27] Srinivas Chellappa, Frédéric de Mesmay, Jared C. Smolens, Babak Falsafi, James C. Hoe, and Ken Mai. Fingerprinting across on-chip memory interconnects. In *Proceedings of the 3rd IEEE Workshop on Silicon Errors in Logic – System Effects (SELSE-3)*, Apr 2007. (poster).

[28] Jone F. Chen, Jiang Tao, Peng Fang, and Chenming Hu. Performance and reliability comparison between asymmetric and symmetric LDD devices and logic gates. *IEEE Journal of Solid-State Circuits*, 34(3), March 1999.

[29] SELSE Organizing Committee. Selse-II reverie. In *Proceedings of the 2nd Workshop on System effects of Logic Soft Errors (SESLE-2)*, Apr 2006.

[30] Compaq Computer Corporation. *Alpha 21264 Microprocessor Hardware Reference Manual*, July 1999.

[31] Zarka Cvetanovic. Performance analysis of the Alpha 21364-based HP GS1280 multiprocessor. In *Proceedings of the 30h Annual International Symposium on Computer Architecture*, pages 218–229, June 2003.

[32] Edward W. Czeck and Daniel P Siewiorek. Effects of transient gate-level faults on program behavior. In *Digest of Papers 20th Annual International Symposium on Fault-Tolerant Computing (FTCS'90)*, Jun 1990.

[33] Dan Ernst, Nam Sung Kim, Shidhartha Das, Sanjay Pant, Rajeev Rao, Toan Pham, Conrad Ziesler, David Blaauw, Todd Austin, Krisztian Flautner, and Trevor Mudge. Razor: A low-power pipeline based on circuit-level timing speculation. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 36)*, December 2003.

[34] Keith I Farkas, Norman P. Jouppi, and Paul Chow. Register file design considerations in dynamically scheduled processors. Technical Report 95/10, Digital Western Research Laboratory, Nov 1995.

[35] Stephen Fischer. Technical overview of the 45nm next generation intel core microarchitecture (penryn). In *Intel Developer Forum*, Apr 2007.

[36] M. Franklin. A study of time redundant fault tolerant techniques for superscalar processors. In *Proceedings of IEEE Intl. Workshop on Defect and Fault Tolerance in VLSI Systems*, 1995.

[37] Simcha Gochman, Avi Mendelson, Alon Haveh, and Efraim Rotem. Introduction to intel core duo processor architecture. Technical report, Intel, 2006.

[38] Simcha Gochman, Ronny Ronen, Ittai Anati, Ariel Berkovits, Tsvika Kurts, Alon Naveh, Ali Saeed, Zeev Sperber, and Robert C. Valentine. The intel pentium m processor: Microarchitecture and performance. In *Intel Technology Journal*, volume 7, May 2003.

[39] Brian T. Gold, Jangwoo Kim, Jared C. Smolens, Eric S. Chung, Vasilis Liaskovitis, Eriko Nuvitadhi, Babak Falsafi, James C. Hoe, and Andreas G. Nowatzyk. TRUSS: a reliable, scalable server architecture. *IEEE Micro*, 25:51–59, Nov-Dec 2005.

[40] Mohamed Gomaa, Chad Scarbrough, T. N. Vijaykumar, and Irith Pomeranz. Transient-fault recovery for chip multiprocessors. In *Proceedings of the 30h Annual International Symposium on Computer Architecture*, June 2003.

[41] Brian Greskamp and Josep Torrellas. Paceline: Improving single-thread performance in nanoscale CMPs through core overclocking. In *Proceedings of the 16th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, Sept 2007.

[42] Richard Hankins, Trung Diep, Murali Annavaram, Brian Hirano, Harald Eri, Hubert Nueckel, and John P. Shen. Scaling and characterizing database workloads: Bridging the gap between research and practice. In *Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 37)*, pages 151–162, Dec 2003.

[43] C-K. Hu, D Camaperi, S. T. Chen, and et al. Effects of overlayers on electromigration reliability improvement for cu/low k interconnects. In *42nd Annual International Reliability Physics Symposium (IRPS)*, 2004.

[44] Intel Corporation. *IA-32 Intel Architecture Software Developer's Manual, Volume 1: Basic Architecture*, 2004.

[45] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*, May 2007.

[46] Intel Corporation. *Intel Core 2 Duo Processors and Intel Core 2 Extreme Processors for Platforms Based on Mobile Intel 965 Express Chipset Family*, Aug 2007.

[47] International Telecommunications Union. *ITU-T X.25: Data Networks and Open System Communication*, 1997.

[48] T. Juhnke and H. Klar. Calculation of the soft error rate of submicron cmos logic circuits. *IEEE Journal of Solid State Circuits*, 30(7):830–834, July 1995.

[49] Jeffrey W. Kellington, Ryan McBeth, Pia Sanda, and Ronald N. Kalla. IBM POWER6 processor soft error tolerance analysis using proton irradiation. In *Proceedings of the 3rd IEEE Workshop on Silicon Errors in Logic – System Effects (SELSE-3)*, Apr 2007.

[50] Kee Sup Kim, Rathish Jayabharathi, and Craig Carstens. SpeedGrade: an RTL path delay fault simulator. In *Proceedings of the 10th Annual Asian Test Symposium*, Nov 2001.

[51] Meyrem Kirman, Nevin Kirman, and José F. Martínez. Cherry-MP: correctly integrating checkpointed early resource recycling in chip multiprocessors. In *Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 38)*, Dec 2005.

[52] Bernd Koenemann, Joachim Much, and Gunther Zwiehoff. Built-in logic block observation techniques. In *Proceedings of the 1979 IEEE Test Conference*, September 1979.

[53] Poonacha Kongetira, Kathirgamar Aingaran, and Kunle Olukotun. Niagara: A 32-way multithreaded sparc processor. *IEEE Micro*, 25(2):21–29, Mar-Apr 2005.

[54] Smita Krishnaswamy, Igor L. Markov, and John P. Hayes. When are multiple gate errors significant in logic circuits? In *Proceedings of the 2nd Workshop on System effects of Logic Soft Errors (SESLE-2)*, April 2006.

[55] Ravishankar Kuppuswamy, Peter DesRosier, Derek Feltham, Rehan Sheikh, and Paul Thadikaran. Full hold-scan systems in microprocessors: Cost/benefit analysis. *Intel Technology Journal*, 8(1), February 2004.

[56] Christopher LaFrieda, Engin İpek, José F. Martínez, and Rajit Manohar. Utilizing dynamically coupled cores to form a resilient chip multiprocessor. In *International Conference on Dependable Systems and Networks*, June 2007.

[57] Shih-Chang Lai, Shih-Lien Lu, Konrad Lai, and Jih-Kwon Peir. Ditto processor. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, Jun 2002.

[58] Yung-Huei Lee, Neal Mielke, Marty Agostinelli, Sukirti Gupta, Ryan Lu, and William McMahon. Prediction of logic product failure due to thin-gate oxide breakdown. In *Proceedings of the 44th Annual International Reliability Physics Symposium*, June 2006.

[59] Yanjing Li, Samy Makar, and Subhasish Mitra. CASP: concurrent autonomous chip self-test using stored test patterns. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE'08)*, 2008.

[60] Barry P. Linder, James H Stathis, David J Frank, Salvatore Lombardo, and Alex Vayshenker. Growth and scaling of oxide conduction after breakdown. In *Proceedings of the 41st Annual International Reliability Physics Symposium*, August 2003.

[61] Jose Maiz, Scott Hareland, Kevin Zhang, and Patrick Armstrong. Characterization of multi-bit soft error events in advanced srams. In *IEEE International Electron Devices Meeting (IEDM)*, Dec 2003.

[62] Deborah T. Marr, Subrananian Natarajan, Shreekant Thakkar, and Richard Zucker. Multiprocessor validation of the pentium pro. In *IEEE Computer*, Nov 1996.

[63] Marthon Technologies Corporation. *Marathon everRun FT whitepaper*, 2007.

[64] Cameron McNairy and Rohit Bhatia. Montecito: A dual-core, dual-thread Itanium processor. *IEEE Micro*, 25(2), March 2005.

[65] Joseph W. McPherson. Reliability challenges for 45nm and beyond. In *Proceedings for the 43rd Annual Design Automation Conference (DAC)*, June 2006.

[66] Patrick J. Meaney, Scott B. Swaney, Pia N. Sanda, and Lisa Spainhower. IBM z990 soft error detection and recovery. *IEEE Trans. device and materials reliability*, 5(3):419–427, Sept 2005.

[67] Avi Mendelson and Neeraj Suri. Designing high-performance and reliable superscalar architectures: The Out of Order Reliable Superscalar O3RS approach. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, Jun 2000.

[68] MIPS Technologies Inc. *MIPS R10000 Microprocessor User's Manual*, version 2.0 edition, 1996.

[69] Subhasish Mitra and Kee Sup Kim. X-Compact: an efficient response compaction technique. *IEEE Transactions on Computer-aided Design of Integrated Circuits and Systems*, 23(3):421–432, March 2004.

[70] Subhasish Mitra, Ming Zhang, Saad Wagas, Norbert Seifert, Balkaran Gill, and Kee Sup Kim. Combinational logic soft error correction. In *Proceedings of the 2006 International Test Conference (ITC)*, Oct 2006.

[71] Shubhendu S. Mukherjee, Joel Emer, and Steven K. Reinhardt. The soft error problem: An architectural perspective. In *Proceedings of the Eleventh IEEE Symposium on High-Performance Computer Architecture (HPCA)*, Feb 2005.

[72] Shubhendu S. Mukherjee, Michael Kontz, and Steven K. Reinhardt. Detailed design and evaluation of redundant multithreading alternatives. In *Proceedings of the 29th Annual International Symposium on Computer Architecture (ISCA)*, May 2002.

[73] Shubhendu S. Mukherjee, Christopher T. Weaver, Joel Emer, Steven K. Reinhardt, and Todd Austin. A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 36)*, Dec 2003.

[74] Suriyaprakash Natarajan, Srinivas Patil, and Sreejit Chakravarty. Path delay fault simulation on large industrial designs. In *Proceedings of the 24th IEEE VLSI Test Symposium*, August 2006.

[75] Umesh Gajanan Nawathe, Mahmudul Hassan, Lynn Warriner, King Yen, Bharat Upputuri, David Greenhill, Ashok Kumar, and Heechoul Park. An 8-core, 64-thread, 64-bit, power efficient SPARC SoC (Niagara 2). In *Proceedings of the International Solid-State Circuits Conference (ISSCC)*, Feb 2007.

[76] Umesh Gajanan Nawathe, Mahmudul Hassan, Lynn Warriner, King Yen, Bharat Uputuri, David Greenhill, Ashok Kumar, and Heechoul Park. An 8-core, 64-thread, 64-bit power

efficient sparc soc (niagara2). In *Proceedings of the International Solid-State Circuits Conference (ISSCC)*, Mar 2007.

[77] Kunle Olukotun, Basem A. Nayfeh, Lance Hammond, Ken Wilson, and Kun-Yung Chang. The case for a single-chip multiprocessor. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VII)*, Oct 1996.

[78] Bipul C. Paul, Kunhyuk Kang, Haldun Kufluoglu, Muhammad A. Alam, and Kaushik Roy. Negative bias temperature instability: Estimation and design for improved reliability of nanoscale circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 26(4), Apr 2007.

[79] Vera Pless. *Introduction to the Theory of Error-Correcting Codes*. Wiley-interscience, 2nd edition, 1989.

[80] Irith Pomeranz, Sandip Kundu, and Sudhakar M. Reddy. On output response compression in the presence of unknown output values. In *Proceedings of the 39th Annual Design Automation Conference (DAC)*, June 2002.

[81] Zach Purser, Karthik Sundaramoorthy, and Eric Rotenberg. Slipstream processors: Improving both performance and fault tolerance. In *Proceedings of the 33rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 33)*, Dec 2000.

[82] Nhon Quach. High availability and reliability in the Itanium processor. *IEEE Micro*, 20(5), Sep-Oct 2000.

[83] Nhon Quach. Private TRUSS group presentation, Oct 3 2006.

[84] Tenkasi V. Ramabadran and Sunil S. Gaitonde. A tutorial on CRC computations. *IEEE Micro*, 8(4):62–75, Aug 1988.

[85] Joydeep Ray, James C. Hoe, and Babak Falsafi. Dual use of superscalar datapath for transient-fault detection and recovery. In *Proceedings of the 34th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 34)*, Dec 2001.

[86] Sudhakar M. Reddy, Kewal K. Sluja, and Mark G. Karpovsky. A data compression technique for built-in self test. *IEEE Transactions on Computers*, 37(9):1151–1156, September 1988.

[87] Vijay Reddy, Anant T. Krishnan, Andrew Marshall, John Rodriguesz, Sreedhar Natarajan, Tim Rost, and Srikanth Krishnan. Impact of negative bias temperature instability on digital circuit reliability. In *40th Annual International Reliability Physics Symposium (IRPS)*, 2002.

[88] Steven K. Reinhardt and Shubhendu S. Mukherjee. Transient fault detection via simultaneous multithreading. In *Proceedings of the 17th Annual International Symposium on Computer Architecture (ISCA)*, Jun 2000.

[89] R. Rodriguez, J. H. Stathis, and B. P. Linder. Modeling and experimental verification of the effect of gate oxide breakdown on CMOS inverters. In *41st Annual International Reliability Physics Symposium (IRPS)*, Aug 2003.

[90] Eric Rotenberg. AR-SMT: A microarchitectural approach to fault tolerance in microprocessors. In *Digest of Papers 29th Annual International Symposium on Fault-Tolerant Computing (FTCS'99)*, Jun 1999.

[91] Franz X. Ruckerbauer and Greorg Georgakos. Soft error rates in 65mn srams - analysis of new phenomena. In *Proceedings of the 13th International On-Line Testing Symposium (ILOTS)*, July 2007.

[92] Stefan Rusu and Simon Tam. Clock generation and distribution for the first IA-64 microprocessor. In *Proceedings of the International Solid-State Circuits Conference (ISSCC)*, 2000.

[93] Kewal. K. Saluja and M. Karpovsky. Testing computer hardware through data compression in space and time. In *Proceedings of the International Test Conference*, pages 83–88, 1983.

[94] John P. Shen and Mikko H. Lipasti. *Modern processor design: fundamentals of superscalar processors*. McGraw Hill, 2005.

[95] Larry Sherman. Stratus continuous processing technology – the smarter approach to uptime. Technical report, Stratus Technologies, 2003.

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

[97] Premkishore Shivakumar, Michael Kistler, Stephen W. Keckler, Doug Burger, and Lorenzo Alvisi. Modeling the effect of technology trends on the soft error rate of combinational logic. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, Jun 2002.

[98] Smitha Shyam, Kypros Constantinides, Sujay Phadke, Valeria Bertacco, and Todd Austin. Ultra low-cost protection for microprocessor pipelines. In *Proceedings of the Twelfth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XII)*, Oct 2006.

[99] Alex Shye, Tipp Moseley, Vijay Janapa Reddi, Joseph Blomstedt, and Danieal A. Connors. Using process-level redundancy to exploit multiple cores for transient fault tolerance. In *International Conference on Dependable Systems and Networks*, June 2007.

[100] Daniel P. Siewiorek and Robert S. Swarz (Eds.). *Reliable Computer Systems: Design and Evaluation*. A K Peters, 3rd edition, 1998.

[101] T.J. Slegel, R.M. Averill III, M.A. Check, B.C. Giamei, B.W. Krumm, C.A. Krygowski, W.H. Li, J.S. Liptay, J.D. MacDougall, T.J. McPherson, J.A. Navarro, E.M. Schwarz, K. Shum, and C.F. Webb. IBM's S/390 G5 microprocessor design. *IEEE Micro*, 19(2):12–23, Mar-Apr 1999.

[102] Gordon L. Smith. Model for delay faults based upon paths. In *Proceedings of the 1983 International Test Conference (ITC)*, Oct 1983.

[103] James. E. Smith and Andrew R. Pleszkun. Implementing precise interrupts in pipelined processors. *IEEE Transactions on Computers*, C-37(5):562–573, May 1988.

[104] Jared C. Smolens, Brian T. Gold, Babak Falsafi, and James C. Hoe. Reunion: Complexity-effective multicore redundancy. In *Proceedings of the 39th ACM/IEEE International Symposium on Microarchitecture*, December 2006.

[105] Jared C. Smolens, Brian T. Gold, Jangwoo Kim, Babak Falsafi, James C. Hoe, and Andreas G. Nowatzyk. Fingerprinting: Bounding soft-error detection latency and bandwidth. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 224–234, Oct 2004.

[106] Jared C. Smolens, Jangwoo Kim, James C. Hoe, and Babak Falsafi. Efficient resource sharing in concurrent error detecting superscalar microarchitectures. In *Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 37)*, December 2004.

[107] E.S. Sogomonyan, A. Morosov, M. Gossel, A. Singh, and J. Rzeha. Early error detection in systems-on-chip for fault-tolerance and at-speed debugging. In *Proceedings of the 19th VLSI Test Symposium*, May 2001.

[108] Daniel J. Sorin, Milo M. K. Martin, Mark D. Hill, and David A. Wood. SafetyNet: improving the availability of shared memory multiprocessors with global checkpoint/recovery. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, June 2002.

[109] Jayanth Srinivasan, Sarita V. Adve, Pradip Bose, and Jude A. Rivers. The case for lifetime reliability-aware microprocessors. In *Proceedings of the 31st Annual International Symposium on Computer Architecture (ISCA)*, June 2004.

[110] Jayanth Srinivasan, Sarita V. Adve, Pradip Bose, and Jude A. Rivers. The impact of technology scaling on lifetime reliability. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, June 2004.

[111] Jayanth Srinivasan, Sarita V. Adve, Pradip Bose, and Jude A. Rivers. Exploiting structural duplication for lifetime reliability enhancement. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture (ISCA)*, June 2005.

[112] Stratus Technologies. *Benefit from Stratus Continuous Processing Technology*, July 2007.

[113] Sun Microsystems. *OpenSPARC T1 Microarchitecture Specification, Revision A*, Aug 2006.

[114] Karthik Sundaramoorthy, Zachary Purser, and Eric Rotenberg. Slipstream processors: improving both performance and fault tolerance. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IX)*, November 2000.

[115] T. N. Vijaykumar, Irith Pomeranz, and Karl Cheng. Transient fault recovery using simultaneous multithreading. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, May 2002.

[116] Mathys Walma. Pipelined cyclic redundancy check (CRC) calculation. In *Proceedings of the 16th International Conference on Computer Communications and Networks*, Aug 2007.

[117] Nicholas J. Wang, Justin Quek, Todd M. Rafacz, and Sanjay J. Patel. Characterizing the effects of transient faults on a high-performance processor pipeline. In *Proceedings of the 2004 International Conference on Dependable Systems and Networks (DSN)*, June 2004.

[118] Thomas F. Wenisch, Roland E. Wunderlich, Michael Ferdman, Anastassia Ailamaki, Babak Falsafi, and James C Hoe. SimFlex: statistical sampling of computer system simulation. *IEEE Micro*, 26(4):18–31, Jul-Aug 2006.

[119] Kent Wilken and John P. Shen. Continuous signature monitoring: Low-cost concurrent detection of processor control errors. *IEEE Transactions on Computer-Aided Design*, 9(6):629–641, June 1990.

[120] Huiyang Zhou. A case for fault tolerance and performance enhancement using chip multiprocessors. In *IEEE Computer Architecture Letters*, Sept 2005.

[121] J. F. Ziegler, et al. IBM's experiments in soft fails in computer electronics (1978-1994). *IBM Journal of Research and Development*, 40(1), 1998.