Sim-SODA: A Unified Framework for Architectural Level Software Reliability Analysis

Xin Fu
Department of ECE
University of Florida
xinfu@ufl.edu

Tao Li
Department of ECE
University of Florida
taoli@ece.ufl.edu

José A. B. Fortes
Department of ECE
University of Florida
fortes@acis.ufl.edu

Abstract

*Semiconductor transient faults (soft errors) are becoming an increasingly critical threat to reliable software execution. With the advent of the billion transistor chip era, it is impractical to protect the entire hardware. As a result, it is crucial that the tradeoffs between reliability and performance be made at the architecture design stage. To achieve this goal, researchers need a framework to evaluate software vulnerability to transient errors at a high level. This paper describes Sim-SODA (**SO**ftware **D**ependability **A**nalysis), a unified framework for estimating microprocessor reliability in the presence of soft errors at the architectural level. Compared with previous studies, Sim-SODA covers more hardware structures and provides fine-grained reliability analysis. We present a detailed architectural reliability profile of an Alpha-21264-like superscalar microprocessor running workloads from various application domains. Additionally, we obtain program vulnerability phases and correlate them with microprocessor performance metrics.*

1. Introduction

High availability and reliability are essential for any computer system. It is well known that program bugs and administration time account for the majority of system downtime and loss of availability. Recently, semiconductor transient faults have become an increasing cause of failures in modern computer systems [2, 27, 28]. Transient faults, also known as *soft errors*, are caused by cosmic rays or substrate alpha particles that can potentially alter program run-time states. As semiconductor processing technology moves toward smaller and denser transistors, lower threshold voltages and tighter noise margins, soft error rates of current and future hardware are projected to increase significantly [15, 21, 23].

Methodologies to tolerate the deleterious effect of soft errors on program execution at low levels exist. For example, radiation-hardening techniques can be used in circuit and logic designs to reduce the likelihood of a single event upset due to soft errors [7]. In [30], SMT capability is exploited to execute a redundant thread to tolerate faults in the main thread. Nevertheless, these approaches can cause a significant overhead in performance, power, area, and design verification if they are used to protect the entire hardware which may contain billions of transistors. Architectural techniques to increase software reliability under soft errors are becoming imperative. To make appropriate performance, cost, and reliability trade-offs; designers clearly need infrastructures that can estimate microprocessor dependability[1] at a high level and at an early design stage. Such tools can be very useful in identifying structures with a high vulnerability and to apply appropriate fault tolerance mechanisms to minimize performance and cost overhead. Using architectural level dependability estimation tools, designers can compare the reliability of different architectural alternatives. Additionally, architectural level simulation tools can help programmers explore the design space of reliability-aware software and verify that a given program execution meets the dependability target on a hardware platform.

This paper describes *Sim-SODA* (***SO***ftware ***D***ependability ***A***nalysis), an architectural level simulator for software reliability analysis. *Sim-SODA* estimates the dependability of hardware components in a high-performance, out-of-order superscalar microprocessor using the computation methods introduced in [4, 22]. Compared with previous studies [4, 19, 22], *Sim-SODA* provides fine-grained reliability analysis and covers more hardware structures. While previous architectural reliability analysis tools were built on proprietary performance models [13, 24], *Sim-SODA* uses an open source, publicly available simulator Sim-Alpha [11, 12], which makes porting the reliability analysis framework described in this paper to other popular simulator tool suites (such as Simplescalar[6] and M5 [3]) relatively easy.

1.1. Prior Work

There has been prior work on dependability modeling at a high level. For example, hardware RTL models have been used in the past to estimate processor reliability [25, 27]. The RTL models contain all of the detailed information about the microprocessors. Nevertheless, the simulation slowdown of RTL models is too expensive for architecture studies, in which the tradeoffs between many hardware configurations need to be considered. Moreover, these models are generally not available during the architectural exploration phase of a microprocessor design. The Architectural Vulnerability Factor (AVF) analysis methods proposed by Mukherjee et al used a performance model to generate reliability estimates. In [4, 22] the vulnerability of hardware structures (e.g. instruction queue, execution unit, TLB and caches) of an Itanium2-like IA64 processor was studied. In [1], Asadi et al estimated the vulnerability of L1 cache through the residency time of critical words in the cache. In [19], Li and Adve developed SoftArch, an architecture level tool for modeling and analyzing soft errors. The SoftArch framework estimates reliability using a probabilistic model of the error generation

[1] In this paper, we use dependability and reliability interchangeably.

Table 1. A Comparison of Different Architectural Level Reliability Analysis Tools

Metrics	Mukherjee et al[22] Biswas et al [4]	Wang et al [27]	SoftArch [19]	Sim-SODA [this paper]
Methodology	AVF	Statistic fault injection	Probabilistic model of error generation and propagation	AVF, AVF for address-based structures and hybrid AVF computing
Hardware Structures Modeled	Instruction queue, function unit, Data cache and TLB, store buffer	Pipeline and its control states	Instruction buffer, decode unit, register file, functional unit, TLB, instruction queue	Instruction queue, register file, function unit, cache, TLB, ROB, load/store queue, victim buffer
Baseline Models and Availability	Asim, Intel's proprietary tool for modeling Itanium 2-like processor	Verilog model of an Alpha processor, available at http://www.crhc.uiuc.edu/ACS/tools/ivm/download.html	Turandot, available on request	Sim-Alpha, publicly available
Comment	Complex hardware such as instruction queue is modeled as bulk structure	A subset of Alpha ISA is modeled. Caches are not modeled. RTL model is not usually available at early design stage	Memory hierarchy is not modeled. Complex hardware such as instruction queue is modeled as bulk structure	Fine-grained AVF models for complex structures. Covers more hardware structures

and propagation process in a processor. As a complementary approach to AVF computation, statistic fault injection has been used in several studies [5, 10, 25, 27] to evaluate architectural reliability. To obtain statistic significance, a large number of experiments need to be performed on an investigated hardware component. Table 1 summarizes the features of several architectural reliability estimation tools from the perspectives of methodology, modeled hardware structures and the availability of baseline models.

1.2. Contribution of This Work

We have developed *Sim-SODA*, a unified simulation framework that models software reliability on microprocessor-based systems. Compared with prior studies, this work makes the following contributions: (1) *Sim-SODA* covers more hardware structures (e.g. reorder buffer and victim buffer) that have not been studied before. While the AVF of microarchitecture structures, such as instruction queue, cache and load/store buffers have been individually studied before, *Sim-SODA* provides a unified infrastructure to study the reliability of all major units of a high-performance microprocessor with a single run. (2) In *Sim-SODA*, we propose using a fine-grained reliability analysis to improve the accuracy of AVF estimation. We also propose a hybrid method that can be used to accurately estimate the vulnerability of complex structures such as register files. To our knowledge, AVF of register files has not been well understood in previous publications. (3) Our work characterizes run-time hardware vulnerability dynamics and its correlation with performance. We show that using a simple performance metric is not sufficient in capturing hardware vulnerability. This observation has not been made in previous studies. (4) *Sim-SODA* was built on the open source and publicly available Sim-Alpha simulator while all other architectural reliability simulators were built on proprietary frameworks. We show that Sim-SODA will be a very useful tool for reliability-aware software/hardware design and optimization.

The rest of this paper is organized as follows. Section 2 provides a brief introduction of reliability estimation through AVF computing. Section 3 describes the design of the *Sim-SODA* framework, especially focusing on the new features that we added. Section 4 presents experimental setup including simulated machine configuration and studied workloads. Section 5 provides a detailed, component-based reliability profile of an Alpha-21264-like microprocessor running on a wide range of applications. Section 6 summarizes the paper and outlines our future work.

2. Architectural Level Software Vulnerability Estimation

Sim-SODA estimates microprocessor reliability using the Architectural Vulnerability Factor (AVF) computing methods introduced in [4, 22]. In this section, we briefly review the concept and the computation of AVF.

Since not all soft errors can cause erroneous program execution, the probability that a fault in a hardware structure will cause an externally visible error in the final output of a program is referred to as the architectural vulnerability factor (AVF) of that hardware structure. A hardware structure's error rate is the product of its raw error rate, mainly determined by device and circuit design technology, and the AVF. The key to calculating the AVF is to determine which bits affect the final system output and which do not. In [22], a subset of processor state bits required for architecturally correct execution (ACE) are called ACE bits. Hence, the AVF of a hardware structure in a given cycle is the percentage of the ACE bits in that structure. The AVF of a hardware structure during program execution is the average AVF at any point in time.

3. The *Sim*-SODA Reliability Estimation Framework

3.1. Overview

We have developed *Sim-SODA*, an architectural framework to estimate the reliability of programs running on high-performance, out-of-order microprocessors. To track the residence time of ACE bits in various structures, we instrumented Sim-alpha, an open source, validated cycle-accurate performance simulator for Alpha 21264. In the *Sim-SODA* framework, we classify each dynamic instruction of a programs execution based on whether the instruction's output affects the outcome of that program. Since instructions executed along a mispredicted path will not be committed, and do not affect AVF, we only considered committed instructions. We consider an instruction an ACE instruction if its results might affect the final program output, and an instruction un-ACE if its results definitely will not affect the program output. Bits in an ACE instruction are ACE, but an un-ACE instruction contains both ACE and un-ACE bits (details are explained in [22]). Additionally, we classify one type of un-ACE instruction dynamically dead if its results are not used subsequently (more detailed classification of un-ACE instructions will be introduced in section 5.1). In *Sim-SODA*, we implemented the post-commit analysis window proposed in [22] to determine if the instruction is dynamically dead or if there are any bits that are logically masked. Through cycle-level simulation, both microarchitecture and architecture states are classified into ACE/un-ACE bits and their residency and resource usage counts are generated. This information is then used to estimate the reliability of various hardware structures.

3.2. Fine-grained Reliability Estimation

3.2.1. Instruction Window

In high performance processors, the instruction window is used to support dynamic scheduling and out-of-order execution. In [22], the instruction window is treated as a bulk structure. *Sim-SODA* provides fine-grained reliability analysis for the instruction window. When an instruction completes its execution, its destination register ID is broadcasted to all the instructions in the window to inform all dependent instructions of the availability of the result. Each entry compares the broadcasted register ID with its own source register ID. If there is a match, the source operand is latched and the dependent instruction may be ready to execute. This register ID broadcast and associated comparison is called instruction wake-up. A soft error that results in an incorrect match between a broadcasted physical register ID and a corrupted tag may cause instructions waiting for that operand to be issued pre-maturely. A single bit error in the tag array that results in a mismatch where there should have been a hit can prevent ready instructions from being issued, causing a deadlock in issuing instructions. Therefore, the wake-up table is vulnerable to soft error strikes. The *Sim-SODA* framework estimates the vulnerability of both the instruction window and the wake-up table.

When a new instruction is allocated in the instruction window, the wake-up table records the renamed physical register IDs of instructions on which that instruction depends. There are two fields in each wake-up table entry to hold the renamed register IDs for the two source operands of an instruction. A field in the wake-up table entry becomes invalid once the source operand is ready. The operations on the wake-up table include “fill”, “read” and “invalidate”; therefore, its non-overlapping lifetime can be partitioned into fill-to-read, read-to-read and invalidate-to-fill periods. Note that there is no read-to-invalidate component because the last read between fill and invalidate will cause a match between the stored register ID and the broadcasted register ID. Once there is a match, the field in the wake-up table will become invalid immediately. In other words, the lifetime of the read-to-invalidate component in the wake-up table is always zero. Therefore, we combine fill-to-read and read-to-read components together, and attribute the invalidate-to-fill component as un-ACE.

3.2.2. Trivial Instruction

In [22], Mukherjee et al identified logical masking instructions as a source of un-ACE bits. An operand and its bits are logically masked and can be attributed to un-ACE bits if the operand does not influence the result of an instruction execution. In their study, Mukherjee et al considered three types of logical masking instructions: compare instructions prior to a branch, bitwise logical operations and 32-bit operations in a 64-bit architecture. In this study, we identified further logical masking bits. We found that the bits used to encode the specifiers of source registers which hold logically masked values are un-ACE bits. This is because a corrupted register specifier may cause the processor to fetch the wrong data from a different register. Nevertheless, the computation result will not be altered because of the logical masking effect.

Additionally, we extend logical masking instructions to trivial instructions [29] in this study. Trivial instructions are those computations whose output can be determined without performing the computation, so they cover all the un-ACE bits that logical masking instructions can identify. In this study, we further classified the trivial instructions into the following three categories. The first type of trivial instructions has two source registers. For these trivial instructions, a soft error is tolerant when it strikes a register whose contribution to the computation result is masked by the second register. For example, in a multiplication instruction, if one of the source registers is equal to zero, a soft error that hits the other register would not affect the result. Therefore, the bits held by that source register are un-ACE bits. Additionally, the bits used for encoding the other source register specifier within the same instruction are also un-ACE bits. The second type of trivial instructions contains an immediate value and only one source register. The bits in the source registers can be considered un-ACE when the immediate value masks the instructions contribution to the computation results. Similarly, bits in the immediate value can be considered un-ACE when the source register doesn’t affect the computation result. The source register specifier bits in that instruction become un-ACE if the value held in that register is un-ACE. Note that in both types of trivial instructions, only one of the source operands can be considered un-ACE at a time, since a soft error hit to the operand which provides the masking function will skew the computation results. The last type of trivial instructions is specific to XOR and EQV operations (see Table 2). For these two operations, when the first and second register specifiers are identical, bits in that register are un-ACE. Table 2 summarizes the trivial instructions identified by the *Sim-SODA* framework. The particular source operand value that trivializes the operation is also listed.

Integer adds and subtractions are not included in Table 2. This is because their computation results depend on both operands. A bit change in either operand may result in incorrect computation output. Integer divisions are not listed in Table 2 because a division operation is implemented through multiplication instructions in Alpha instruction set [8, 9, 16]. Note that trivial instructions can be dynamically dead instructions also. If an instruction is both a trivial instruction and a dynamically dead instruction, we attribute it to the dynamically dead instructions because more un-ACE bits can be derived from that type of instruction. In other words, trivial instructions analyzed by the *Sim-SODA* framework are special ACE instructions that have un-ACE bits in the instructions and their registers.

Table 2. Trivial Instructions (in Alpha ISA) identified by *Sim-SODA*

Type	Operation	Triviality Condition
I	MULL/MULQ/MULH: A * B	A=0 or B=0
	AND: A & B	A=0 or B=0
	BIS: A | B	A=1 or B=1
	BIC: A & ~ B	A=0 or B=1
	ORNOT: A | ~ B	A=1 or B=0
II	MULLI/MULQI/MULHI/: A * IMM	A=0 or IMM=0
	ANDI: A & IMM	A=0 or IMM=0
	BISI: A | IMM	A=0 or IMM=1
	BICI: A & ~ IMM	A=0 or IMM=1
	ORNOTI: A | ~ IMM	A=0 or IMM=0
III	XOR: A ^ B	A=B
	EQV: A ^ ~B	A=B

3.3. Reliability Estimation of Unexplored Structures

3.3.1. Hybrid AVF Computation for Register Files

The register files hold the architectural state of program execution. Previous studies [19, 25, 27] show that the register files are highly vulnerable to soft errors. In these studies, the reliability analysis of register files was performed by injecting faults statistically or modeling error propagation. The AVF calculation for register files has not been addressed.

When comparing register files with the data cache, we summarize some common characteristics between the two. The register files are similar to the data array in a data cache: both are used to keep values for instruction execution. Activities occurring during the lifetime of a bit in the register files also include "idle", "fill", "read", "write" and "evict". Therefore, if we follow the methodology to compute the AVF for address-based structures [4], we can also classify the register files' lifetimes into non-overlapping ACE or un-ACE periods. For example, idle, read-to-write and write-to-write are un-ACE, while fill-to-read and write-to-read are ACE. However, the calculation of register files' AVF in this way can be very conservative. This is because unlike the data cache, registers are heavily utilized and write and read operations occur very frequently.

To obtain a more realistic estimate of register files' AVF, we analyzed each ACE lifetime component to discover which operations on the bits convert ACE lifetime to un-ACE lifetime. We found that not every read affects the final program output. For example, if there is a read caused by a dynamically dead instruction, the final output will not change even if the data is incorrect. Written data is also un-ACE when the write is caused by a dynamically dead instruction. Additionally, as described in section 3.2.2, when a trivial instruction has a read operation on bits of the register files, the read data is un-ACE if it doesn't affect the result of the instruction. Since we combine ACE/un-ACE lifetime analysis with an instruction analysis window (which is used to detect dynamically dead instructions and trivial instructions, see section 4), we call our method a hybrid AVF computation scheme.

A write to the register file is usually followed by more than one read. Two types of reads can occur. The first is caused by ACE instructions and we call it an ACE read. The second is caused by dynamically dead or trivial instructions and we call it an un-ACE read. Whether to convert ACE lifetime to un-ACE lifetime depends on the order un-ACE reads take place among all the reads after a write. We identified three cases based on the order of un-ACE read operations. First, as Figure 1 (A) shows, un-ACE reads (marked as read*) occur closely after the write activity. The second case is shown in Figure 1 (B). There are one or more ACE reads before and after un-ACE reads. In the third case, as illustrated by Figure 1 (C), there are no more ACE reads but another write or evict follows un-ACE reads. In Figure 1 (A) and (B), un-ACE reads can not be converted from ACE time into un-ACE time since they are followed by ACE reads which can not bear any soft error. In Figure 1 (C) if we follow the methodology applied in address-based structure AVF computation [4], the lifetime component between the last ACE read and the last un-ACE read should be identified as ACE, however, we can convert it into un-ACE since there are no more ACE reads following un-ACE reads. To calculate the un-ACE lifetime, we identify the last ACE read after the write and then attribute the remaining lifetime between it and the next write or evict to un-ACE.

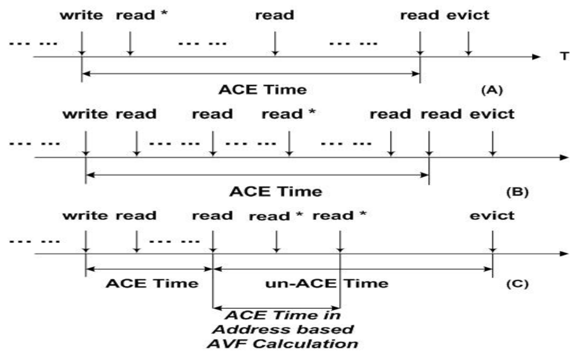
write read * read read evict
T
ACE Time
(A)
write read read read * read read evict
ACE Time
(B)
write read read read * read * evict
ACE Time
un-ACE Time
(C)
ACE Time in Address based AVF Calculation

Figure 1. ACE and un-ACE lifetime partitions due to the different orders of un-ACE reads (marked as read*) and ACE reads

Similar to the data cache array, edge effects also arise in register files' AVF computation. For example, if the simulation ends at a point after a write to the register files completes, we can not determine whether the period between that write and the ending point is ACE lifetime or un-ACE lifetime. Therefore, we have to count the above period as unknown. Since COOLDOWN mechanism [4] has a remarkable impact on reducing the unknown portion of a data cache array's AVF, we applied the COOLDOWN strategy to compute register files' AVF.

The granularity at which we maintain the lifetime information can have a significant impact on register files' AVF. We can't set the granularity to 64-bit; since not all of the instructions defined in the Alpha Instruction set [8, 9, 16] consume the whole 64-bit data word (a register in Alpha 21264 processor is 64-bit). They read or write 32-bit data occasionally, which means the other 32-bits in that register are idle at that time. The methodology used to compute the AVF of address-based structures [4] partitions a tag-based structure into two parts: data array and tag array. Since register files are specified by their identifiers instead of tag array, there is no false positive or false negative match in the register files' AVF computation. Because the case in which read data from register files is less than 32-bit occurs infrequently, it is unnecessary to perform per-byte analysis, or even more detailed per-bit analysis such as the one we used to analyze the logical masking effect. In this study, we maintain granularity for register files' AVF analysis as 32-bit.

3.3.2. ROB AVF Computation

In an out-of-order execution microprocessor, the reorder buffer (ROB) stores all uncommitted instructions. To effectively exploit ILP, modern processors use large ROB, implying its AVF can greatly affect the AVF of the entire chip. We have developed an ROB AVF model for the *Sim-SODA* framework.

Data in each ROB entry is allocated for an on-the-fly instruction. A ROB entry includes instruction number, register specifiers and operands. If an instruction is un-ACE, program output will not be affected, so bits in that entry are un-ACE bits. If an instruction is a trivial instruction, some bits in that entry are un-ACE. In the *Sim-SODA* framework, we use an instruction analysis window and trivial instruction classification to identify these scenarios in the ROB.

3.3.3. Victim Buffer's AVF Computation

The *Sim-SODA* framework models the AVF of a victim buffer. Whenever there is a cache miss in the L1 cache, the replaced block will be evicted to the victim buffer. Since the evicted block may be recalled by the program again, any single bit error in it can cause incorrect program output. The victim buffer is also an address-based structure, and only has "fill", "read", "evict" and "end" activities. We classified no-overlapping ACE, un-ACE and unknown lifetime components on it. For example, fill-to-read, read-to-read are ACE; read-to-evict, fill-to-evict, evict-to-fill, evict-to-end are un-ACE; and fill-to-end, read-to-end are unknown. We used COOLDOWN and hamming distance one mechanisms introduced in [4] to accurately compute AVF.

The above AVF models can be integrated into a range of architectural simulators to provide reliability estimates. To implement these AVF models into a unified, timing accurate framework, we have instrumented the Sim-Alpha architectural simulator. We chose Sim-Alpha because previous work [11, 12] has shown that Sim-Alpha can accurately model an Alpha 21264 processor, and in [11, 12], the authors showed that Sim-Alpha is much more accurate than Simplescalar for modeling real hardware We have extended the simulator with a post-commit instruction analysis window (with a size of 40,000 instructions) which supports the identification of dynamically dead and trivial instructions. The *Sim-SODA* framework also includes AVF models for cache, TLB, and load/store queue. We used synthesized micro-benchmarks with known characteristics to validate the *Sim-SODA* framework. The dynamically dead and trivial instructions and the ACE and un-ACE time reported by *Sim-SODA* match our expectation in the micro-benchmarks.

4. Experimental Setup

Using the *Sim-SODA* framework, we performed a detailed reliability analysis of an Alpha-21264-like microprocessor running a wide range of applications. To help the reader understand the experimental results, we describe the simulated machine configuration and the experimented benchmarks in this section.

4.1. Simulated Machine Configuration

We configured *Sim-SODA* to simulate an Alpha-21264-like microprocessor. Table 3 summarizes the simulated machine configuration.

Table 3. Simulated Machine Configuration

Parameter	Configuration
Pipeline depth	7
Integer ALUs/multi	4/4
Integer ALU/multi latency	1/7
Fetch/slot/map/issue/commit width	4/4/4/4/11 instructions per cycle
Issue queue size	20
Reorder buffer size	80
Register file size	80
Load/store queue size	32
Branch predictor	Hybrid, 4K global + 2-level 1K local+ 4K choice
Return address stack	32-entry
Branch misprediction penalty	7 cycles
L1 instruction cache	64KB instruction/64KB data,2-way, 64B line, 1-cycle latency
L1 data cache	64KB instruction/64KB data,2-way, 64B line, 3-cycle latency
L2 cache	2048KB, direct mapped, 64B line, 7-cycle latency
TLB size	128-entry ITLB/128-entry DTLB, fully-associative
MSHR entries	8/cache
Prefetch MSHR	entries 2/cache
Victim buffer	8 entries, 1-cycle hit latency

4.2. Simulated Workloads

The workloads we used in this study include 12 programs from SPEC 2000 INT and 6 programs from BioInfoMark [20]. We didn't include SPEC 2000 FP benchmarks because Sim-Alpha does not model floating point pipeline execution accurately [11]. To reduce the simulation time while still maintaining representative program behavior, we obtained the number of instructions to skip using SimPoint analysis [26] and run each SimPoint for 50 million instructions. Table 4 lists the skipped instructions and the input data set for each benchmark. The numbers we present in this paper are results for the first SimPoint of each benchmark.

We abbreviate the input names as follows: *bzip2-source* is *bzip2-s*, *gcc-166* is *gcc-1*, *eon-rushmeier* is *eon-r*, *gzip-graphic* is *gzip-g*, *parser-dict* is *parser-d*, *perlbmlk-splitmail* is *perlbmlk-s*, *vpr-route* is *vpr-r*, *clustalw-ureaplasma* is *clustalw-u*, *dnapenny-ribosomal* is *dnapenny-r*, *glimmer-bacteria* is *glimmer-b*, *hmmer-SWISS-PORT* is *hmmer-S*, *predator-eukaryote* is *predator-e*, *promlk-17 species* is *promlk-1*.

Table 4. SPEC 2000 INT and BioInfoMark Benchmarks (The data set name is integrated with the benchmark name)

SPEC 2000 INT	Instructions Fast Forwarded	BioInfoMark	Instruction Fast Forwarded
bzip2-s	64 M	*clustalw*-u	20,400 M
gcc-1	30 M	*dnapenny*-r	140 M
crafty	123 M	*glimmer*-b	20 M
eon-r	216 M	*hmmer*-S	27,200 M
gap	88 M	*predator*-e	25,900 M
gzip-g	1 M	*promlk*-1	320 M
mcf	143 M		
parser-d	1,771 M		
perlbmlk-s	1 M		
twolf	312 M		
vortex-3	47 M		
vpr-r	3 M		

5. Experimental Results

This section presents the architecture vulnerability estimation of the studied workloads running on the simulated machine. Section 5.1 illustrates software vulnerability at the instruction level. Section 5.2 reports vulnerability characterization of major microarchitecture structures. Section 5.3 presents the phase behavior of program vulnerability on various hardware structures and their correlations with program performance statistics.

5.1. Program Vulnerability Profile at Instruction Level

Figure 2 shows an instruction level vulnerability profile of the studied benchmarks. On average, 69% and 73% of the committed instructions are ACE instructions for SPEC 2000 integer and BioInfoMark suites respectively. The un-ACE instructions include NOPs, prefetch and dynamically dead instructions. As suggested in [22], dynamically dead instructions can be classified into two types: (1) first-level dynamically dead (FDD) if their computation results are simply not read by any other instructions, or (2) transitively dynamically dead (TDD) if their results are only consumed by FDD or other TDD instructions. The “Unknown” refers to those instructions whose destination registers’ lifetimes can not be determined by the instruction analysis window.

As shown in Figure 2, NOPs and FDD instructions (FDD_reg and FDD_mem) dominate un-ACE instructions In this study, we found 10% NOPs in the SPEC2000 integer suite. This is the same as the NOPs fraction Fahs et al [14] reported in SPEC2000 integer using the Alpha instruction set. TDD instructions (TDD_reg and TDD_mem) contribute a negligible fraction (e.g. less than 1%) of un-ACE instructions. Similar to the results reported in [22], our study shows that the fraction of FDD_reg (11% in SPEC and 9% in BioInfoMark) instructions is normally higher than that of FDD_mem (7% in both benchmark suites) instructions. The fraction of TDD and FDD instructions reported by *Sim-SODA* framework is 17% in SPEC2000 suite and it is close to that reported in [14] (14% FDD and TDD instructions tracked via register and memory).

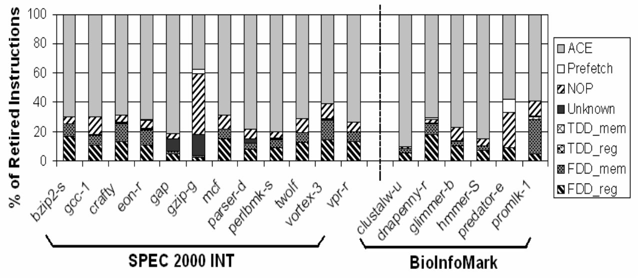

Figure 2. Instruction Level Vulnerability Profile of the Studied Benchmarks

5.2. AVF of Major Microarchitecture Structures7

This section presents AVF profiles of the instruction window, wake-up table, register file, data cache, TLB, victim buffer and load/store queues. Per-structure AVF estimates help hardware designers estimate the reliability of major hardware components at an early design stage.

5.2.1. Instruction Window

Figure 3 shows the AVF of the instruction window. We further decompose ACE bits stored in the instruction window based on their instruction types. As can be seen, the dominant portion of ACE bits in the instruction window comes from ACE instructions. For prefetch and NOP instructions, only instruction opcodes are ACE bits [22]. In this work, we count all the opcode and destination register specifier bits of FDD and TDD instructions as ACE bits; all other instruction bits are un-ACE bits [22].

Figure 3 shows that the instruction window’s AVF ranges from 26% (*gzip*) to 62% (*gap*) in SPEC 2000 and from 44% (*dnapenny*) to 84% (*clustalw*) in BioInfoMark respectively. On average, the AVF of the instruction window is 42% and 52% in SPEC 2000 and BioInfoMark. Biological multiple sequence alignment benchmark *clustalw* has the highest AVF. This is because *clustalw* has the highest ACE instruction fraction (e.g. 90% as shown in Figure 2). Instruction residency time in the instruction window also affects the AVF result, so the benchmark *promlk* yields a higher AVF than *hmmer* even though its ACE instruction fraction is lower than *hmmer* (58% vs. 85% respectively).

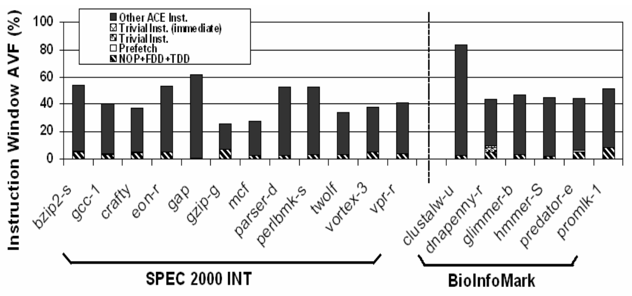

Figure 3. AVF of Instruction Window

Figure 4 shows the AVF of the instruction window, the wake-up table and the aggregated results (i.e. considering the instruction window and the wake-up table as a single structure). We can see the AVF of the wake-up table is much lower than that of the instruction window. This is because an instruction's ACE time in the wake-up table is always shorter than the instruction's residency time in the instruction window. An instruction may still need to wait for a function unit by staying in the instruction window after all of its source operands are ready. As shown in Figure 4, the aggregated results do not reduce significantly (4%-10% in SPEC 2000 and 6%-9% in BioInfoMark). This is because the number of bits contained in the wake-up table is less than that in the instruction window.

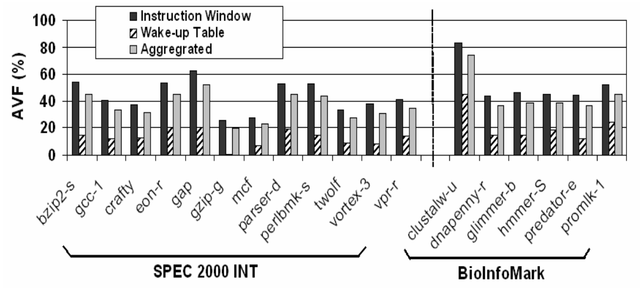

Figure 4. AVF of Instruction Window and Wake-up Table

5.2.2. Reorder Buffer

Figure 5 shows the AVF of the reorder buffer. Interestingly, the ROB's AVF is significantly lower than that of the instruction window. This is due to the following effect. The Alpha-21264 processor has separate integer and floating point instruction windows. The integer instruction window has 20 entries. The ROB is used to hold all types of instructions and the size of the ROB is 80 entries. Because of the lack of floating point operations, the fraction of idle bits in the ROB is much higher than that in the instruction window.

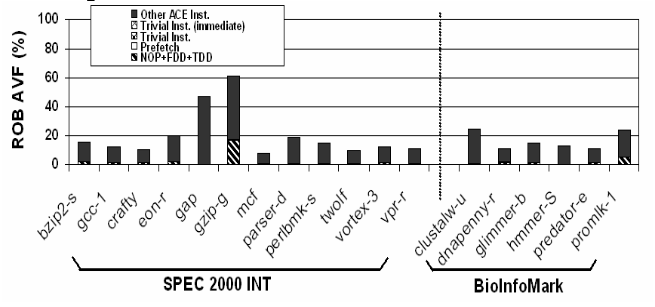

Figure 5. AVF of ROB

5.2.3. Register Files

Sim-SODA models both the high and low 32 bit of the physical registers. In this paper, we report the average AVF of the entire 64-bit registers. As shown in Figure 6, hybrid AVF computation can reduce register files' AVF on many workloads (e.g. 9% on *crafty*, 10% on *promlk* and 13% on *predator*). On average, hybrid AVF calculation reduces register files' AVF by 4% and 5% on SPEC 2000 and BioInfoMark respectively.

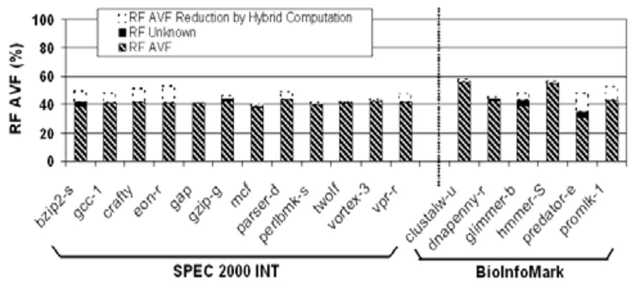

Figure 6. AVF of Register Files

5.2.4. Function Unit

We assume each function unit has about 50% control latches and 50% datapath latches, and the datapath within it has a width of 64 bits. The AVF numbers shown in Figure 7 are the average statistics of all four function units. We apply trivial instruction analysis to each function unit to further attribute un-ACE bits to different instructions. The semantics of trivial instructions implies that at least one input value to the function unit can be un-ACE. There are other instructions that only produce 32-bit outputs. In that case, the upper 32-bits in the output data path become idle. As is shown, function unit AVF bits are mainly caused by the operands of ACE instructions as well as the output of those instructions. Because the majority of instructions have two input operands and one output operand, the input bits contribute more ACE bits in the function units than the output bits do.

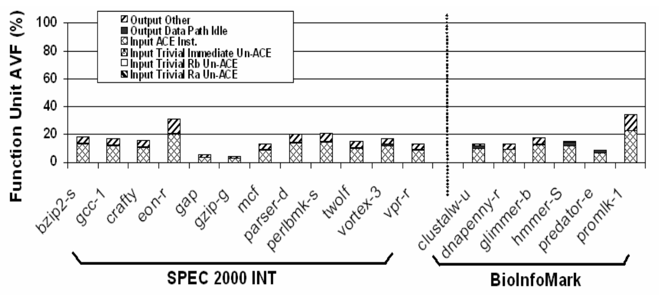

Figure 7. AVF of Function Unit

5.2.5. Data Cache, TLB, Victim Buffer Load and Store Queues

The level 1 data cache, data TLB, victim buffer and load/store queues are address-based structures. We applied lifetime analysis to both tag and data arrays of these structures and classified lifetime into ACE, un-ACE and unknown components. The *Sim-SODA* framework uses bit level analysis for tag array and byte level analysis for data array. We implemented the COOLDOWN mechanism to reduce the unknown fraction since edge effect can be significant in these structures [4]. To avoid false positive and false negative matches in the tag array, we have also implemented the hamming-distance-one analysis method [4] in *Sim-SODA*.

Figure 8 and 9 show the data array and the tag array AVF for L1 data cache (DL1), data TLB (DTLB), victim buffer (VBuf), load queue (LQ) and store queue (SQ). As can be seen, the L1 data cache tag array's AVF is higher than the data array's AVF. This is because the L1 data cache in the Alpha

21264 [17, 18] is a write-back cache and the cache tag must be correct at eviction time. Therefore, all bits of the tag are ACE from the time that there is any write activity that occurs in that entry until it is evicted. The same scenario happens in load and store queues. In contrast, the victim buffer tag array's AVF is lower than the data array's AVF. This is because it is a write-through cache and the ACE time in the tag array is much lower.

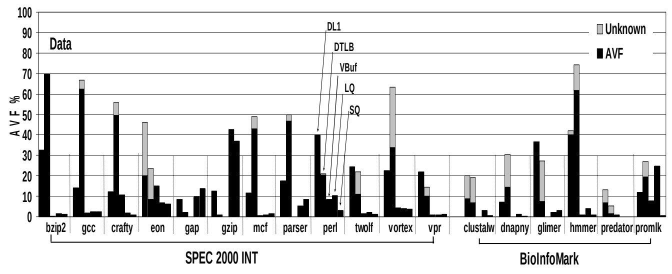

Figure 8. Data Array AVF of L1 Data Cache (DL1), Data TLB (DTLB), Victim Buffer (VBuf), Load Queue (LQ) and Store Queue (SQ)

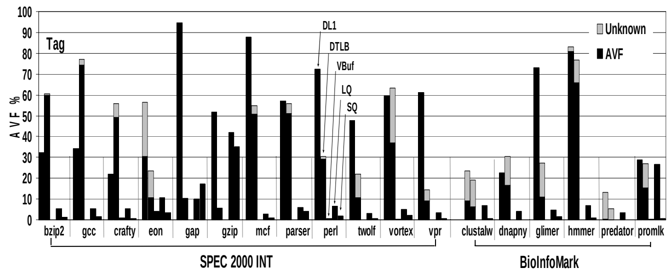

Figure 9. Tag Array AVF of L1 Data Cache (DL1), Data TLB (DTLB), Victim Buffer (VBuf), Load Queue (LQ) and Store Queue (SQ)

5.3. Program Dynamic AVF Behavior

We have performed simulations using *Sim-SODA* to correlate AVF with processor performance statistics such as instructions completed per cycle. We collected the average AVF and performance statistical data for fixed chunks of 10,000 instructions. The sampled vulnerability and performance statistics on benchmarks *gcc* and *gap* are shown in Figure 10.

Figure 10 shows that similar to performance metric IPC, microarchitecture AVF also demonstrates phase behavior. For example, the AVF of instruction window correlates well with IPC on benchmark *gcc* whereas on benchmark *gap*, low IPC phase exhibits high AVF. These statistics were then correlated with each other after the simulation completed. An example of the correlation data for each of the studied benchmarks is shown in Table 5. The columns of this table show correlation coefficients between processor IPC and the AVF of the instruction window, ROB, FU and the wakeup table. Table 5 shows that the wake-up table's AVF correlates very strongly with performance. This is because the lifetime of ACE bits in the wake-up table strongly depends on the number of cycles that the processor completes instructions. In general, we found that the correlation coefficients vary significantly across different benchmarks. This implies that architectural independent characteristics such as fraction of NOPs and dynamically dead instructions in a code segment affect program run-time AVF behavior.

Interestingly, we found that microarchitecture AVF does not always positively correlate with IPC. Intuitively, high IPC reduces the ACE bits residency time in microarchitecture structures. On the other hand, the high ILP in a program can cause the microprocessor to aggressively bring more instructions into the pipeline, increasing the total number of ACE bits. The positive and negative correlation coefficient values imply that AVF is also related with program inherent behavior. The above observations indicate that accurate AVF modeling should consider aspects of both hardware and software. For example, program characteristics of a given code segment (e.g. instruction mix, logical masking instructions) can be combined with hardware performance counters (e.g. pipeline stalls, IPC) to produce accurate AVF estimates for software at run time.

Table 5. Correlation between IPC and AVF

Correlation Coefficient	**AVF**			
	Instruction Window	**ROB**	**FU**	**Wake-up Table**
bzip2-s	0.14	0.18	0.15	-0.13
gcc-1	0.59	0.44	0.27	0.48
crafty	0.11	0.08	0.12	0.64
eon	-0.07	-0.16	0.14	-0.40
gap	-0.77	-0.94	0.94	-0.18
gzip-g	0.53	-0.16	0.72	0.87
mcf	0.69	0.59	0.65	0.99
parser-d	0.22	-0.06	0.45	-0.33
perlbmlk-s	-0.13	-0.32	-0.32	-0.27
twolf	-0.06	-0.03	-0.09	0.73
vortex-3	0.14	-0.02	-0.01	-0.31
vpr-route	0.15	-0.21	0.05	0.63
clustalw-u	0.09	0.76	-0.18	0.78
dnapenny-r	0.12	-0.01	-0.05	0.74
glimmer-b	0.23	-0.04	0.29	0.27
hmmer-S	0.37	0.24	0.01	0.67
predator-e	-0.01	0.07	-0.03	0.99
promlk-1	0.01	-0.28	-0.46	-0.53

6. Conclusions

Semiconductor transient faults have become an increasing challenge for reliable software execution. To explore cost-effective fault tolerant mechanisms for dependable execution of the next generation of software, researchers clearly need to analyze program vulnerability to soft errors at a high level and at an early design stage. We have developed *Sim-SODA*, a unified framework to estimate software vulnerability to transit faults at the architectural level. The foundations for our vulnerability modeling infrastructure are parameterized AVF models of microarchitecture structures present in modern high-performance microprocessors. Compared with previously proposed tools, *Sim-SODA* provides fine-grained AVF models and covers more hardware structures. Using the *Sim-SODA* framework, we profile AVF characteristics of the majority of hardware structures in an Alpha-21264-like microprocessor [17, 18]. Currently, Sim-SODA does not model the vulnerability of the floating point pipeline. Additionally, *Sim-SODA* does not model the AVF contribution of combinational logic due to their negligible impact reported in other studies [22]. Extensions to the simulator infrastructure and the creation of additional modules are topics of future research. We feel reliability estimation infrastructure such as *Sim-SODA* will be useful to research on architecture and compiler approaches to optimize software dependability.

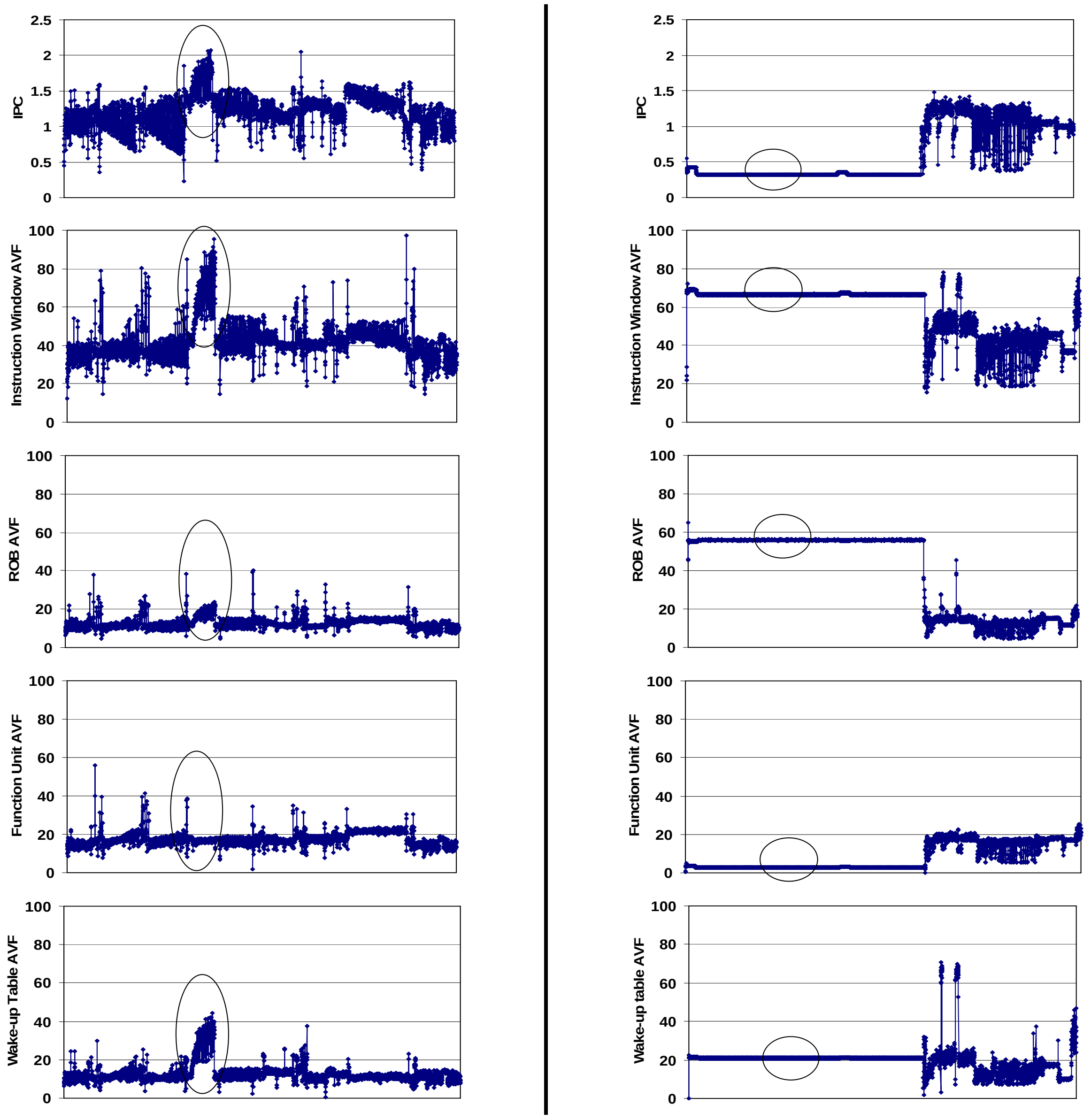

Figure 10. Microarchitecture AVF Dynamics on Benchmarks *gcc* (left) and *gap* (right)

Acknowledgment

This work is supported by the NASA award no. NCC 2-1363 and by the Microsoft Research Trustworthy Computing award no. 14707.

References

[1] G. Asadi, V. Sridharan, M. B. Tahoori, and D. Kaeli, Balancing Performance and Reliability in the Memory Hierarchy, In Proceedings of the International Symposium on Performance Analysis of Systems and Software, 2005.

[2] R. C. Baumann, Soft Errors in Commercial Semiconductor Technology: Overview and Scaling Trends, In IEEE 2002 Reliability Physics Tutorial Notes, Reliability Fundamentals, 2002.

[3] N. L. Binkert, E. G. Hallnor, and S. K. Reinhardt, Network-Oriented Full-System Simulation using M5, In Proceedings of the Workshop on Computer Architecture Evaluation using Commercial Workloads, 2003.

[4] A. Biswas, R. Cheveresan, J. Emer, S. S. Mukherjee, P. B. Racunas, and R. Rangan, Computing Architectural

Vulnerability Factors for Address-Based Structures, In Proceedings of the International Symposium on Computer Architecture, 2005.

[5] J. Blome, S. Mahlke, D. Bradley, K. Flautner, A Microarchitectural Analysis of Soft Error Propagation in a Production-Level Embedded Microprocessor, In Proceedings of Workshop on Architectural Reliability, 2005.

[6] D. Burger and T. M. Austin, The SimpleScalar Tool Set, Version 2.0, University of Wisconsin-Madison, Computer Science Dept., Technical Report No. 1342, June 1997.

[7] T. Calin, M. Nicolaidis, and R. Velazco, Upset Hardened Memory Design for Submicron CMOS Technology, IEEE Transactions on Nuclear Science, Vol. 43, No. 6, Dec. 1996.

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

[9] Compaq Computer Corporation, Compiler Writer’s Guide for the Alpha 21264, 1999.

[10] K. Constantinides, S. Plaza, J. Blome, B. Zhang, V. Bertacco, S. Mahlke, T. Austin, and M. Orshansky, Assessing SEU Vulnerability via Circuit-Level Timing Analysis, In Proceedings of the Workshop on Architectural Reliability, 2005.

[11] R. Desikan, D. Burger, S. W. Keckler, and T. Austin, Sim-alpha: A Validated, Execution-Driven Alpha 21264 Simulator, Technical Report, TR-01-23, Dept. of Computer Sciences, University of Texas at Austin, 2001.

[12] R. Desikan, D. Burger, and S. W. Keckler, Measuring Experimental Error in Microprocessor Simulation, In Proceedings of the Annual International Symposium on Computer Architecture, 2001.

[13] J. Emer, P. Ahuja, N. Binkert, E. Borch, R. Espasa, T. Juan, A. Klauser, C.-K. Luk, S. Manne, S. S. Mukherjee, H. Patil, and S. Wallace, Asim: A Performance Model Framework, IEEE Computer, 35(2):68-76, Feb. 2002.

[14] B. Fahs, S. Bose, M. Crum, B. Slechta, F. Spadini, T. Tung, S. J. Patel, and S. S. Lumetta, Performance Characterization of a Hardware Mechanism for Dynamic Optimization, In Proceedings of the International Symposium on Microarchitecture, 2001.

[15] T. Karnik et al., Characterization of Soft Errors Caused by Single Event Upsets in CMOS Processes. IEEE Trans. Dependable and Secure Computing, 1(2):128–143, June 2004.

[16] R. E. Kessler, E. J. McLellan, and D. A. Webb, The Alpha 21264 Microprocessor Architecture, In Proceedings of the International Conference on Computer Design, 1998.

[17] R. E. Kessler, The Alpha 21264 Microprocessor, IEEE Micro, Vol. 19, No. 2, page 24-36, March/April, 1999.

[18] D. Leibholz and R. Razdan, The Alpha 21264: A 500 MHz Out-of-Order Execution Microprocessor, Proc. Compcon, pp.28-36, 1997.

[19] X. D. Li, S. V. Adve, P. Bose, and J. A. Rivers, SoftArch: An Architecture Level Tool for Modeling and Analyzing Soft Errors, In Proceedings of the International Conference on Dependable Systems and Networks, 2005.

[20] Y. Li, T. Li, T. Kahveci, J. A. B. Fortes, Workload Characterization of Bioinformatics Applications, In Proceedings of the International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, 2005.

[21] S. Mitra, N. Seifert, M. Zhang, Q. Shi, and K. S. Kim, Robust System Design with Built-In Soft-Error Resilience, Computer, Vol. 38, No. 2, page 43-52, Feb. 2005.

[22] S. S. Mukherjee, C. Weaver, J. Emer, S. K. Reinhardt, and T. Austin, A Systematic Methodology to Compute the Architectural Vulnerability Factors for a High-Performance Microprocessor, In Proceedings of the International Symposium on Microarchitecture, 2003.

[23] H. T. Nguyen and Y. Yagil, A Systematic Approach to SER Estimation and Solutions, In Proceedings of the 41st IEEE International Reliability Physics Symposium, 2003.

[24] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August, SWIFT: Software Implemented Fault Tolerance, In Proceedings of the International Symposium on Code Generation and Optimization, 2005.

[25] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, D. I. August, and S. S. Mukherjee, Design and Evaluation of Hybrid Fault-Detection Systems, In Proceedings of the International Symposium on Computer Architecture, 2005.

[26] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, Automatically Characterizing Large Scale Program Behavior, In Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems, 2002.

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

[28] C. Weaver, J. Emer, S. S. Mukherjee, and S. K. Reinhardt, Techniques to Reduce the Soft Error Rate of a High-Performance Microprocessor, In Proceedings of the International Symposium on Computer Architecture, 2004.

[29] J. J. Yi and D. J. Lilja, Improving Processor Performance by Simplifying and Bypassing Trivial Computations, In Proceedings of the IEEE International Conference on Computer Design, 2002.

[30] S. K. Reinhardt and S. S. Mukherjee, Transient Fault Detection via Simultaneous Multithreading, In Proceedings of the International Symposium on Computer Architecture, 2000.