

# Examining ACE Analysis Reliability Estimates Using Fault-Injection

Nicholas J. Wang  
Department of Electrical and  
Computer Engineering  
University of Illinois  
Urbana-Champaign  
nwang@crhc.uiuc.edu

Aqeel Mahesri  
Department of Electrical and  
Computer Engineering  
University of Illinois  
Urbana-Champaign  
mahesri@crhc.uiuc.edu

Sanjay J. Patel  
Department of Electrical and  
Computer Engineering  
University of Illinois  
Urbana-Champaign  
sjp@crhc.uiuc.edu

## ABSTRACT

ACE analysis is a technique to provide an early reliability estimate for microprocessors. ACE analysis couples data from abstract performance models with low level design details to identify and rule out transient faults that will not cause incorrect execution. While many transient faults are analyzable in ACE analysis frameworks, some are not. As a result, ACE analysis is conservative and provides a lower bound for the reliability of a processor design. Bounding the reliability of a design is useful since it can guarantee that the given design will meet reliability goals.

In this work, we quantify and identify the sources of ACE analysis conservatism by comparing an ACE analysis methodology against a rigorous fault-injection study. We evaluate two flavors of ACE analysis: a “simple” analysis and a refined analysis, finding that even the refined analysis overestimates the soft error vulnerability of an instruction scheduler by 2-3x. The conservatism stems from two key sources: from lack of detail in abstract performance models and from what we term Y-Bits, a result of the single-pass simulation methodology that is typical of ACE analysis. We also examine the efficacy of applying ACE analysis to a class of “partial coverage” error mitigation techniques. In particular, we perform a case study on one such technique and extrapolate our findings to others.

## Categories and Subject Descriptors

B.8.1 [Performance and reliability]: [Reliability, Testing, and Fault-Tolerance]

## General Terms

Measurement, Reliability, Experimentation

## Keywords

Fault tolerance, soft errors, measurement techniques, microprocessors

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ISCA '07, June 9–13, 2007, San Diego, California, USA.

Copyright 2007 ACM 978-1-59593-706-3/07/0006 ...\$5.00.

## 1. INTRODUCTION

As processor implementation technologies scale into the deep submicron regime, the issue of controlling the impact of transient errors becomes more and more prevalent. Transient errors can arise from multiple sources: external sources such as high-energy particles that cause current pulses in digital circuits, as well as internal sources that include coupling, leakage, power supply noise, and process variations.

While transient errors (also known as soft errors) have always to some extent plagued semiconductor-based digital systems, the scaling of devices, operating voltages, and design margins for purposes of performance and functionality raises concerns about the increased susceptibility of future-generation systems to such transient effects. Historically, transient errors were primarily of concern to those designing high-availability systems or systems used in electronics-hostile environments such as outer space. Because of the confluence of device and voltage scaling and the increasing complexity of digital systems, the problem of transient errors is forecasted to be a problem for all future digital systems. From high-energy neutrons alone, experts estimate that Failures in Time (FITs) for a chip will scale at a minimum with the number of devices (i.e., with Moore’s Law).

Processor architects require a tool to accurately and quickly estimate the reliability of their designs so that they can make high level architectural trade-offs early in the design process. Traditional methods like fault injection into a hardware description of a processor or radiation testing on a physical device are not available early enough in the design process to close the feedback loop.

ACE analysis [6] was conceived to tackle this problem. ACE analysis utilizes a high-level performance model coupled with low level information about a processor design to provide an early reliability estimate. Furthermore, ACE analysis provides a lower bound on the reliability of a design. Such bounds are useful in demonstrating that a design will meet reliability goals, and they are especially useful if the bounds are sufficiently tight or if reliability goals are lax or otherwise easily met.

A key trade-off made in ACE analysis is a lower degree of accuracy when compared to more detailed fault analysis methodologies. This is due to conservatism forced by maintaining a lower bound. In this paper, we examine the conservatism in ACE analysis for estimating soft error rates of processor systems. We find that the conservatism stems from two key sources: (1) a lack of ACE analysis detail (and

the difficulty associated with increasing the level of detail) and (2) fundamental limitations as a result of utilizing a single-pass simulation to examine all faults.

A lack of detail is a significant contributor to ACE analysis conservatism. In Sections 3 and 4, we provide two data points to illustrate the effect that ACE analysis detail has upon reliability estimates. First, we show that a “simple” ACE analysis (that is nonetheless faithful to the state of the art in the literature) overestimates soft error vulnerability by a factor of 3.5x when compared to fault injection. We then perform a “refined” ACE analysis by increasing the level of analysis detail (and thus simulation complexity) and observe the improvement in accuracy. As we see in Section 4.1, the overestimation remains large with a factor of 2.6x.

ACE analysis’ use of single-pass simulation also presents limitations. Some transient faults dramatically alter the course of execution in a processor, yet do not affect architecturally correct execution (i.e. they are unACE and do not cause a soft error). We term these single bit-flip faults Y-Bits. In similar fashion to Y-Branches [11], Y-Bits are dynamic instances of bits of state whose values are unACE. Examples of Y-Bits may include dynamic instances of instruction opcode bits, valid bits, and head and tail queue pointers. Since ACE analysis typically uses a single simulation to analyze all of the state of a processor at once, it is difficult to perfectly analyze the ACE-ness of bits in this set. As a result, some additional conservatism is introduced. We examine this effect in Section 4.2.

We also observe that ACE analysis has difficulty providing accurate reliability estimates for a certain class of fault tolerant processor architectures: those that feature “partial” fault mitigation techniques and whose placement of fault coverage is often affected by the presence of soft errors. ACE analysis has difficulty with this class of architectures for two reasons: (1) the conservatism described previously leads to a poor distinction between masked faults and soft errors and (2) the lack of detail and the single-pass simulation methodology make it difficult to identify when and where fault coverage is active. In this work, we empirically evaluate the efficacy of an ACE analysis of one such architecture: Re-Store [12], a cost effective fault tolerant architecture that features symptom based soft error detection. Then we generalize our findings to examine the effect of applying ACE analysis to other partial coverage architectures.

Despite these limitations, early stage techniques such as ACE analysis are indispensable for estimating the fault tolerance of a processor, particularly because ACE analysis provides a lower bound on the reliability of a design early in the design process. The results of this study help bridge the gap between the estimates provided by ACE analysis and the actual fault tolerance of a design.

## 2. BENCHMARK FAULT ANALYSIS

To perform an analysis of the conservatism introduced by ACE analysis, we require a benchmark methodology to compare against. To be meaningful, the benchmark methodology should be as precise as possible. To this end, we elected to use a fault injection methodology on a low level hardware description of a processor coupled with detailed analysis of the processor and workload’s behavior after injection.

To be clear, we are not presenting an argument that fault

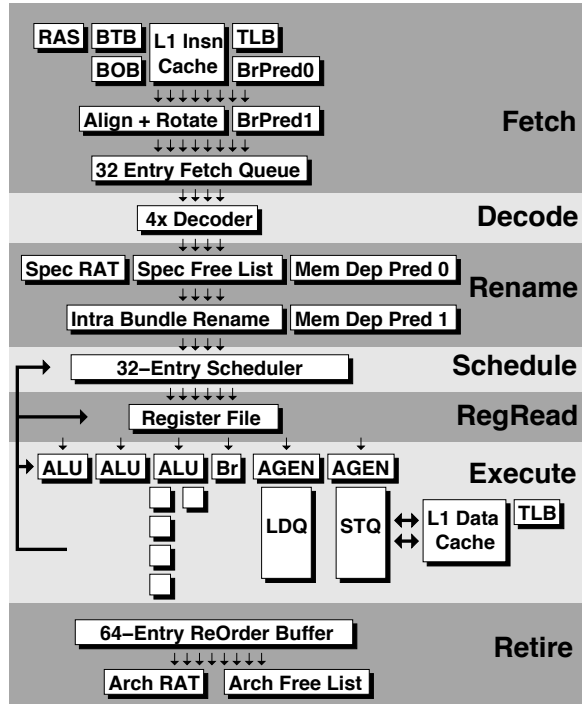


Figure 1: IVM processor diagram.

injection should be utilized in place of ACE analysis. Both analysis techniques have advantages and disadvantages and neither can replace the other. Instead, we are simply using one strength of fault injection (accuracy of analysis) to identify where and how approximations are made in ACE analysis methodologies.

What follows is a description of the fault injection and analysis methodology that we will use as a benchmark in the remainder of the paper.

### 2.1 Methodology

We performed our fault injection experiments on IVM [13] (Illinois Verilog Model), a Verilog implementation of an Alpha microprocessor. IVM models a superscalar, dynamically scheduled pipeline. The modeled processor includes such features as speculative instruction scheduling, memory dependence prediction, and sophisticated branch prediction. Up to 132 instructions can be in-flight in the 12-stage pipeline. The processor core consists of approximately 40,000 bit elements of state (pipeline latches and RAM storage) in addition to caches and predictor tables. The architecture implemented in IVM is presented in Figure 1.

Prior to performing the fault injection, the caches and predictor tables are warmed up from the beginning of the workload. We used seven SPEC2000 integer benchmarks as workloads: bzip2, cc1, gap, gzip, mcf, parser, and vortex. After the caches and predictor tables have been warmed up, the remainder of the processor is warmed up by simulating for approximately 100,000 cycles. At this point, the processor is in steady state execution and is ready for fault injection.

We used a single bit-flip in microarchitectural state as our fault model. At the fault injection clock cycle, a single bit of

microarchitectural state is selected at random, and its value is inverted. Caches were not fault injected because they are easily protected by parity or ECC. We also excluded predictor tables from fault injection because faults in predictor tables are always masked from propagating to software visible state. All other microarchitectural state is eligible for fault injection. Examples include pipeline state, register files, and data and instruction buffers in the microarchitecture.

After fault injection, we carefully observe the propagation of the bit corruption as the processor proceeds to execute the workload. The processor is monitored for deadlocks/livelocks and instruction set architecture defined exceptions. If either of these events are detected, the injected fault is declared to have propagated to a DUE (detected unrecoverable error) event. Note that simply detecting a deadlock or exception is not typically sufficient to classify an error as DUE (as opposed to SDC). Instead, one must also be able to show that there were no silent and persistent data corruptions (e.g. corruptions silently written to another application’s memory space or propagated to disk) prior to the DUE event. Our simulation environment consists of a single workload, and we aggressively assume that any file corruptions are ignored or overwritten after the process is terminated. This greatly simplifies SDC/DUE analysis, possibly introducing error into distinguishing between SDC and DUE events but not for overall reliability estimates.

In the absence of exceptions or deadlocks, the processor is allowed to run up to 10,000 cycles past the fault injection. At this point, if both the architectural state image of the workload and the microarchitectural state of the processor are clean, then the fault is known to be benign (masked). If the architectural state image of the workload is corrupt, we assume that all microarchitectural effects have completely transferred to architectural state and proceed to simulate the workload in a virtual machine environment.

Cases exist where the architectural state image of the workload is clean, but the microarchitectural state of the processor is not. In these cases, after 10,000 simulation cycles the fault remains latent in the microarchitectural state. It is possible that these faults will be activated sometime in the future and cause an SDC or DUE event. Manual inspection of these cases revealed that the vast majority are benign — most faults in this class were injected into seldom used queue entries and had not been overwritten before the 10,000 cycle limit was reached.

After the corrupt architectural state has been transferred to a virtual machine environment, the workload is simulated to completion. At this point, manual inspection of the workload output reveals whether the injected fault was benign, DUE, or SDC. Examples of workload output that place a fault into the DUE category include programmer assertion fires, CRC check failures (prevalent in bzip2 and gzip), and other obviously incorrect output (for example, no output at all!). If the program output is corrupted in a meaningful way, absent of any DUE conditions, we label the fault SDC. Otherwise, the fault is benign. A flow chart depicting the fault injection and analysis process is provided in Figure 2.

Each of the experimental results presented in this work represent about 10,000 fault injections, yielding a 1% confidence interval at a 95% confidence level. In other words, if an experiment were to be performed 20 times, 19 of the ex-

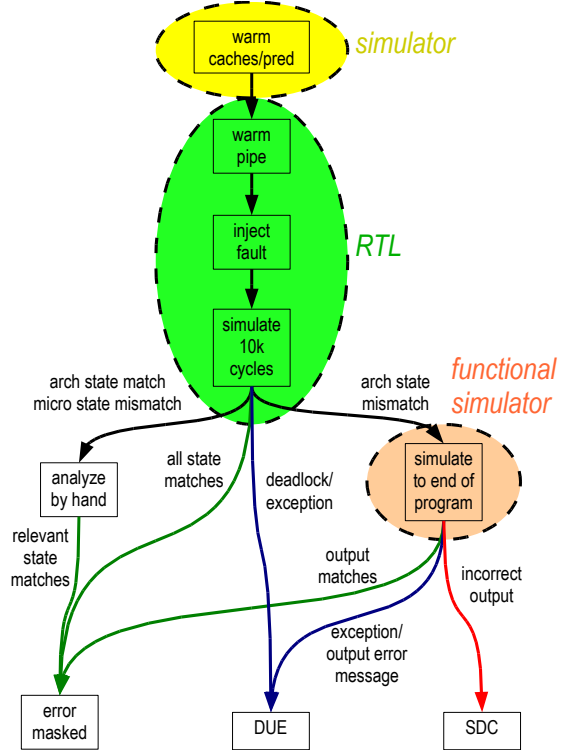


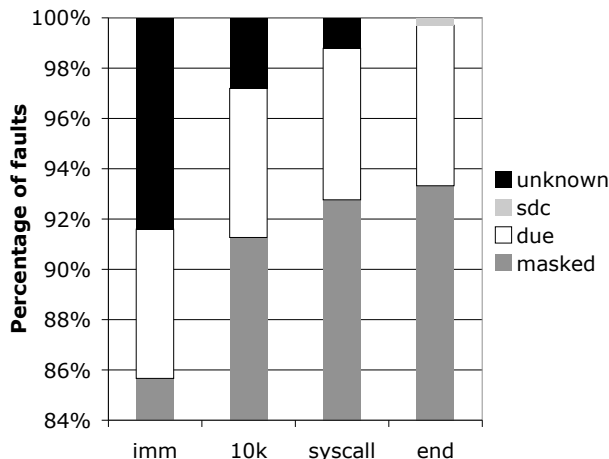
Figure 2: Flow chart of fault injection and analysis process.

perimental results are expected to fall within 1% of the true value. Although it is often desirable to have as small an error margin as possible, the accuracy of the absolute experimental result is not the key metric in this study. In this work, we compare the results from two analysis methodologies. Thus the difference in analysis outcome for each trial between the two compared methods is what is most important.

## 2.2 Result accuracy versus cooldown length

Before we present the bulk of our experimental data, we examine the benefits of the fault injection and analysis methodology presented in Section 2.1. In particular, we examine the trade-off between result accuracy and cooldown [1] length, a measure of the length of simulation after each transient fault used for fault analysis. Figure 3 presents data from a fault injection campaign using the methodology described in the previous subsection. The y-axis spans all fault injections while four different bars span the x-axis. From left to right, the four bars represent snapshots of estimated processor reliability obtained from increasing cooldown length: immediately after detection of any software visible error, after 10,000 processor cycles have elapsed since fault injection, at the first system call, and at program termination. Each of the four bars is composed of four categories: masked/benign faults, DUE, SDC, and unknown.

At the *imm* cooldown length, we stop the RTL simulation immediately upon deadlocks, exceptions, and architectural state corruptions. In the absence of any of these events, the simulation is allowed to proceed for up to 10,000 cycles. Deadlocks and exceptions are treated as DUE events while



**Figure 3: Cooldown length versus estimated processor reliability.**

architectural state corruptions automatically place the fault into the “unknown” category. Simulation results after the architectural state corruption are ignored, so the ultimate outcome is unknown. About 86% of faults are conclusively masked while 8% are unknown. This methodology is similar to the one used in prior work [13] that isolated the effects of microarchitectural and software level masking.

At the *10k* cooldown length, deadlocks and exceptions are still treated as DUE events when appropriate. However, in the absence of DUE, the fault is simulated for a full 10k cycles. If the architectural state of the processor is consistent at that point, the trial falls into the *masked* category, regardless of whether or not an architectural state corruption occurred previously. By taking this additional step, 5% of previously *unknown* trials are identified as *masked*. This methodology is similar to the one used in prior work to evaluate the ReStore architecture [12].

In the *syscall* configuration, *unknown* trials from *10k* are allowed to simulate in a virtual machine up until the next system call. In the *end* configuration, *unknown* trials are allowed to simulate until the end of the workload. Going from the *10k* configuration to *end*, an additional 2% of trials move from *unknown* to *masked*. Out of 10,550 fault injections, 673 resulted in DUE while 32 resulted in SDC. Note that no parity was implemented anywhere in the processor to convert SDC to DUE!

As faults are simulated for longer periods of time, more information becomes available about the fault’s effects. Note that in the *end* configuration, zero trials are left in the *unknown* category. We observe that after a microarchitectural fault has propagated to architectural state, most of the masking effects can be captured by simulating for 10,000 cycles after the fault injection or until the next system call.

The vast majority of soft errors are DUE — although this result is highly dependent on the workload in question. We observed that *bzip2*, *gzip*, and *mcf* were the only SDC contributors in our benchmark suite. All 321 errors observed in *gcc*, *gap*, *parser*, and *vortex* were DUE. *Bzip2* and *gzip* have CRC in their decompression stages to catch most soft errors and convert them to DUE. However, their compres-

sion stages are rather vulnerable to SDC events since any corrupted output data is not discovered until the CRC is calculated and compared during decompression. In similar fashion to the decompression stages in *bzip2* and *gzip*, software assertions in *gap* and *parser* act to significantly increase the proportion of DUE errors.

### 3. ACE ANALYSIS

Now that we have established a reference fault analysis methodology, we can proceed to compare it against ACE analysis.

#### 3.1 ACE Analysis Overview

The ACE analysis methodology was conceived as a method to provide an early estimate of the response of a processor architecture to transient faults. This goal was accomplished by first noting that there are many obvious cases when structures in a processor are not sensitive to data corruptions. For example, when a scheduler entry is not occupied by an instruction, nearly all of its state is immune to transient faults. Bits of state that must be correct for correct execution of the program are called ACE (Architecturally Correct Execution), and bits of state that are tolerant of data corruption are called unACE [6]. When a scheduler entry is not occupied by an instruction, nearly all of the state of that entry is unACE.

ACE analysis utilizes a performance timing model to track instructions as they move through the processor pipeline and occupy various pipeline structures. With careful analysis of the structures, it is possible to identify which bits of state within the structure are unACE when unoccupied. Combined with structure occupation data from the performance timing model, it is simple to estimate its vulnerability factor — or the percentage of time that an incident transient fault leads to error.

ACE analysis estimates are calculated such that they create a conservative bound on the vulnerability factor. Only unACE cases are identified — in other words, only bits that are positively known to be unACE are labeled as such. All other bits are conservatively assumed to be ACE. One way to tighten this bound is to analyze the instruction stream to identify dynamically dead instructions and then treat the dead instructions differently while they occupy the structures in question. Processor state occupied by misspeculated instructions can be treated similarly. This process helps to tighten the reliability estimate because knowing that a dead instruction is occupying a structure often yields enough information to identify a greater portion of state as unACE.

ACE analysis is not limited to analysis of structures that queue instructions. Its philosophy can be applied towards tag and data stores as well [1]. This is accomplished by analyzing the access patterns (again, supplied by a performance model) of the structure in question.

A key advantage of ACE analysis is derived from its ability to use data generated by performance models instead of low level hardware descriptions. Performance models are available early on in the design cycle of a processor, allowing ACE analysis to be performed early enough such that high level trade-offs can still be made between reliability, performance, power, die area, and other metrics. Other techniques to analyze fault tolerance include fault injection on a hard-

ware description of the processor or radiation testing of a physical device. However, by the time the tools required by these techniques are available, it is usually too late for any high level changes to be made. As a result, the processor may be over-designed or under-designed for reliability to the detriment of other metrics.

Unfortunately, ACE analysis is not without its weaknesses. One such weakness is that it generates a necessarily conservative reliability estimate. Another is that there is some state in processors where it is not yet apparent that ACE analysis is applicable. Examples include valid bits, the program counter register, and head and tail pointers for queues. Investigating the sources of these weaknesses are the focus of this work.

### 3.2 Instruction scheduler description

In this study, we will focus solely on the IVM instruction scheduler. Furthermore, we restrict our analysis to the portion of the scheduler that ACE analysis can analyze — namely, everything in the scheduler except (1) a scoreboard state machine that tracks ready operands and (2) the head and tail pointers that respectively denote the youngest and oldest instructions in the queue. We perform an experiment to analyze the omitted state in Section 4.2.

By focusing our investigation on the instruction scheduler, we forfeit lessons that might only be obtained from analyzing other structures. However, we note that the fields in the scheduler payload (see Table 1) mirror the fields found in many other structures throughout our processor. Thus, the lessons learned from examining the scheduler can be generalized and applied towards a large portion of the processor pipeline. Portions of modern processors where this is not true include datapaths and address-based structures. We note that prior work [1, 6] has observed lower bounds on the soft error vulnerability of these structures, decreasing their significance in terms of our analysis of conservatism.

Before proceeding further, we will describe the inner workings of the IVM instruction scheduler and the criteria that we selected for identifying ACE bits in the ACE analysis. Table 1 itemizes and describes the contents of each of the 32 entries in the IVM instruction scheduler.

Our ACE analysis algorithm utilizes knowledge of whether each scheduler entry is occupied (valid) or not. Furthermore, if the entry is occupied, it is known whether the instruction is live, dead, or wrong path. With this information, we classify each bit of state into one of three categories: (1) *Always* - bits that are always ACE, (2) *Valid* - bits that are ACE only when the corresponding hardware is an instruction (regardless of live, dead, or wrong path status), and (3) *Valid and live* - bits that are ACE only when a live (and thus necessarily valid) instruction is present. Live instructions are also referred to as ACE instructions. The ACE criteria for each of the fields in the IVM scheduler are given in Figure 1.

In the scheduler, each entry’s valid bit cannot be derated with the information given, so they are classified as always ACE. The program counter and branch unit control fields are only relevant for control instructions. Since all control instructions are live, the program counter and branch unit control fields are classified as *Valid and live*. The remainder of the scheduler state is potentially ACE whenever its corresponding entry is valid and active, so they are placed in the *Valid* category.

### 3.3 Experimental methodology

One of the strengths of ACE analysis is the ability to analyze all the processor state in question in a single simulation pass. In this study, however, we forego this advantage in the interest of performing an apples to apples comparison of ACE analysis against fault injection.

Instead of using a performance model to feed ACE analysis with scheduler utilization data, we extract the utilization data directly from IVM. Instructions that pass through the scheduler are monitored for occupation times as well as for retirement. Knowing that an instruction retires implies that the instruction was correct path. Correct path instructions are then analyzed for liveness.

We analyze each injected fault in isolation from one another. This is normal for fault injection but not for ACE analysis. We perform each fault injection independently so that we can perform a precise analysis of the effect of each fault on the processor model using the methodology described in Section 2. ACE analysis is then performed only on the fault injected bits of state. Thus, comparisons between ACE analysis and fault injection are apples to apples and reveal where and how ACE analysis is conservative.

ACE analysis was performed only on the ACE-analyzable state of the scheduler. Correspondingly, fault injection was also restricted to the same set of state.

### 3.4 ACE analysis

Figure 4 plots the data utilized by the ACE analysis of the scheduler: the percentage of time that the average scheduler entry is occupied by (1) a live instruction (*ace insn*), (2) correct path dead instructions (*unace cp insn*), or (3) any other instruction (*valid insn*). In the remainder of cycles (60%), the scheduler entries sit idle. In IVM’s scheduler implementation, instructions remain for two cycles after they are last issued. The two cycle delay ensures that the instruction will not require replay before it is removed from the scheduler. These two cycles of unACE time are placed in the *valid insn* category along with all cycles attributed to wrong path instructions.

The percentage of dead instructions was 19% of all dynamic instructions, approximately half of which were transitively dead. Transitively dead instructions are dead instructions whose output is consumed only by other dead instructions. Note that the *unace cp insn* category comprises significantly fewer than one-fourth the total number of correct path instruction cycles in the scheduler — less than might be expected given the 19% dead instruction statistic. The reason the *unace cp insn* category is smaller than expected is because many of the dead instructions are NOP’s which are never inserted into the IVM scheduler. As a result, going to the effort of distinguishing between the *Valid and live* and *Valid* categories in Table 1 will have little impact on the final results.

On average, scheduler entries are occupied by ACE instructions about 23% of the time and by unACE instructions 16% of the time. Using this data and the scheduler ACE state breakdown listed in Table 1, we arrive at Figure 5 which plots the masking level of the scheduler, as determined by ACE analysis versus that of fault injection. Each of the bars indicates the percentage of transient faults that are estimated to be masked by the corresponding analysis

Field description	Abbr.	Bits	ACE criteria
Program counter	pc	20	Valid and live
Valid bit	valid	1	Always
Functional unit designator	fud	5	Valid
Functional unit control	fuc	22	Valid
Branch unit control	buc	5	Valid and live
Miscellaneous control	mc	5	Valid
Source register pointers	reg	16	Valid
Destination register pointer	reg	8	Valid
Instruction tag	tag	6	Valid

Table 1: Breakdown of each scheduler entry.

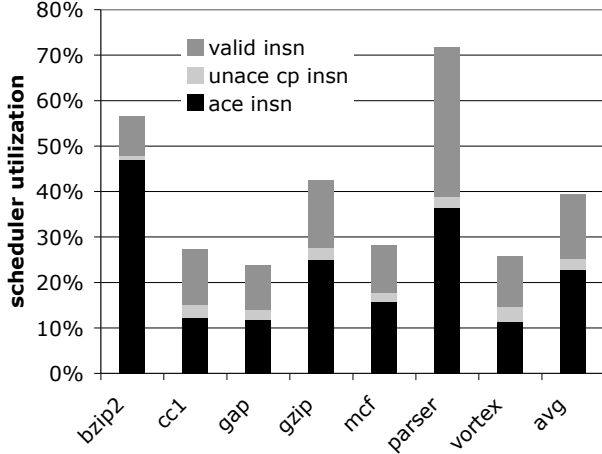


Figure 4: Scheduler entry occupation data.

methodology. The remainder propagate to an SDC or DUE soft error. The *FI* bars depict the result arrived at by fault injection while the *ACE* bars depict the result obtained by performing ACE analysis on the same bit-flips.

Since ACE analysis provides a bound on the true result, the *ACE* bars are always lower than the *FI* bars. In this particular case ACE analysis bounds the level soft error masking of the scheduler to be 69% while the fault injection analysis reports 91%.

There are three sources for this conservatism. The first source of conservatism is from the limited context used to classify each of the bits of state in the scheduler. We will investigate this contributor more closely in Section 4.1. The second source of conservatism is a fundamental limitation of ACE analysis which we term Y-Bits. We discuss and investigate this phenomenon in Section 4.2. The third is that the dead instruction analysis window used in ACE analysis was about 100k instructions, compared to an infinite window in fault injection. This is the *cooldown* [1] period, analogous to the warmup period commonly used in performance simulations. This factor is not likely a key contributor given the data in Figure 3.

### 3.5 ACE analysis of ReStore

ACE analysis can be used to analyze the level of soft error masking present in a processor design. ACE analysis is also useful for providing an early estimate of the fault coverage provided by soft error mitigation techniques. In this

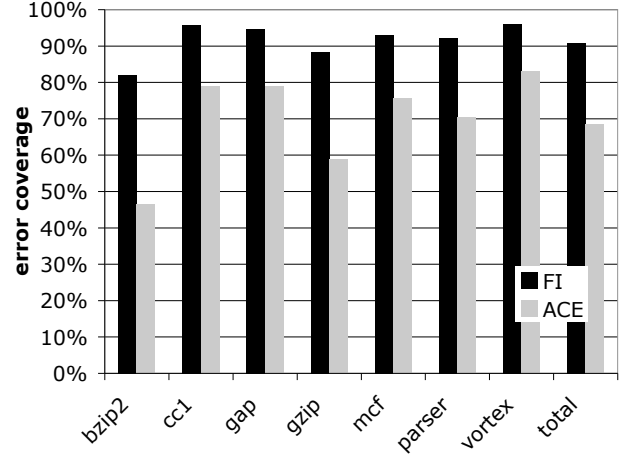


Figure 5: Fault masking in a scheduler: fault injection versus ACE analysis.

section, we explore the limitations of applying ACE analysis to a class of error mitigation techniques: those that feature “partial” fault mitigation and whose placement of fault coverage is often affected by the presence of soft errors.

Researchers have proposed several cost effective methods for obtaining (near) complete coverage [10, 14, 8] as well as partial coverage [15, 3, 12, 7]. Complete coverage is characterized by providing full redundancy such that failure is extremely rare. On the other hand, partial coverage is characterized by somewhat less error coverage, accompanied with the benefit of lower overheads in terms of area, performance, and/or power. In many partial coverage techniques, the placement of fault coverage is driven by certain microarchitectural phenomena, including cache misses and branch mispredictions. If soft errors affect fault coverage placement (perhaps via affecting microarchitectural phenomena), then ACE analysis can have difficulty providing accurate reliability estimates. We will now illustrate these observations with a case study of ReStore.

The ReStore architecture [12] is a low cost error mitigation technique that utilizes soft error *symptoms* to assist error detection. Soft error symptoms are atypical events that often *quickly* accompany a soft error. Note that we distinguish between transient faults and soft errors. Soft errors are transient faults that propagate to a DUE or SDC failure. ReStore detects and recovers from transient faults by maintaining low overhead hardware checkpoints and listen-

ing for symptoms. When a symptom is detected, a prior checkpoint is restored. Re-execution of the intervening instructions enables fault detection (which can be as simple as looking for the same symptom to occur again) while the checkpoint restore enables recovery. The ReStore architecture is low cost because it re-uses existing mechanisms to perform checkpointing and symptom detection.

The ReStore architecture that we elected to evaluate for this study guarantees a minimum rollback distance of 100 instructions when soft error symptoms are detected. It was configured to use pipeline deadlock, exception, high confidence conditional branch mispredictions, and load and store misses to memory as symptoms.

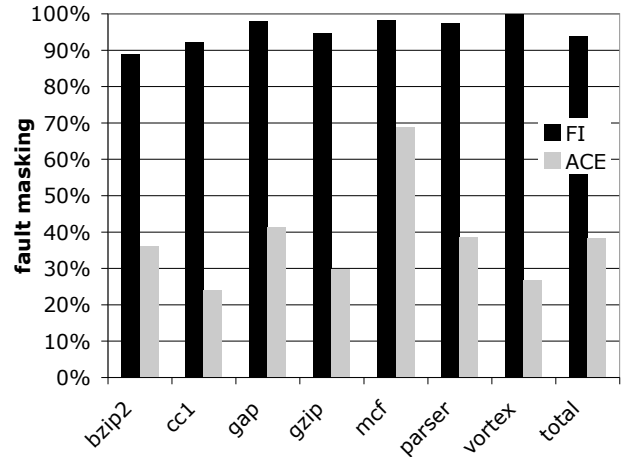
ACE analysis relies upon identifying unACE bits of state. Since most of the state of a processor can be analyzed for unACE-ness in parallel, ACE analysis is typically performed on all eligible processor state in a single simulation run (we will call this *single-pass ACE analysis*). With single-pass ACE analysis, it can be difficult to analyze certain error mitigation techniques. ReStore is one such example.

Evaluation of ReStore requires more detail than is typically provided in performance models. To evaluate ReStore, the propagation of transient faults through the processor must be tracked to determine if they trigger soft error symptoms. If single-pass ACE analysis is used, complexity must be introduced into the simulator to enable this tracking. To this end, recent work [7] utilized fault injection into software state (instruction outputs) to evaluate the reliability of a similar architecture. In this work, we have a different goal and approach the problem differently: we perform fault injection on our low level processor model, extracting symptoms from the fault injection trial and giving ACE analysis oracle knowledge of any soft error symptoms that arise. This process gives ACE analysis the benefit of doubt about identifying ReStore’s error detection and recovery opportunities.

Figure 6 compares the ACE analysis estimation of fault coverage of the scheduler in the ReStore processor versus the fault coverage as reported by the fault injection analysis. The y-axis in the plot spans all of the ACE fault injections in Figure 5 for the corresponding analysis methodology. In other words, for *FI* the y-axis spans the 9% of fault injections that were not masked in Figure 5 and for *ACE*, the y-axis spans the corresponding 31% of trials determined to be ACE by ACE analysis.

The ACE analysis estimate is significantly more conservative. It claims an error coverage figure of about 38%, while the fault injection campaign reports 94%. In other words, the ACE analysis would have us believe that ReStore decreases the soft error vulnerability of the scheduler by about a factor of 1.6x, whereas the fault injection analysis reports a factor of approximately 16x.

While comparing error coverage figures is interesting, the piece of data that truly matters when comparing reliable architectures is the overall estimated reliability. In this case study, ACE analysis is conservative in identifying the masking level of the baseline processor’s scheduler as well as in evaluating the error coverage of ReStore’s scheduler. The net result is that there is a 38x difference between the ACE analysis estimate versus the fault injection derived estimate. The fault injection analysis reported a 0.5% (the baseline processor’s scheduler was 9% vulnerable and ReStore de-



**Figure 6: Error coverage in a ReStore scheduler: fault injection versus ACE analysis.**

fects and recovers from 94% of errors) vulnerability for the ReStore processor’s scheduler while ACE analysis reported 19% (the ACE analysis estimate for the baseline processor’s scheduler was 31% vulnerability while the ReStore estimate was 38%). The 38x difference falls out from examining the difference between 0.5% and 19%.

Many soft error reliability studies have utilized ACE analysis to evaluate proposed mechanisms. In those cases, the error coverage data depicted in Figure 6 is relevant, since the error coverage claimed by a study is a key metric in comparing it to other studies. Note that error coverage claims generated by ACE analysis are not bounded, since coverage is calculated as a ratio between two ACE analysis outputs.

The inaccuracy of the ACE analysis of ReStore is due to two reasons. First, ReStore works by responding to symptoms, which are strongly biased towards faults that result in error (SDC or DUE). As a result, ReStore is biased towards providing error detection only when errors are likely present. Second, ACE analysis is conservative in identifying errors in the baseline processor, labeling many masked faults as errors. These two facts lead to ACE analysis significantly underestimating the error coverage provided by the ReStore architecture.

To summarize, there are two key issues that arise when performing ACE analysis on ReStore. First, ReStore’s redundancy is triggered by events that are not easily captured in typical ACE analysis frameworks. We eliminated this issue in this study by providing ACE analysis with oracle information about symptoms. Second, as noted in the previous paragraph, since ReStore’s placement of redundancy is strongly biased around soft errors, conservatism in identifying soft errors in the baseline architecture leads to significant under-reporting of error coverage.

Other partial coverage error mitigation techniques are similarly affected. For example, in Opportunistic Transient-Fault Detection [3], redundancy is provided whenever economical — for instance, during misses to memory. The authors utilized ACE analysis to evaluate their architecture. Many soft errors cause additional misses to memory [12] which would not be identified in their single-pass and rel-

atively abstract analysis methodology. Thus their reliability estimates may be somewhat inaccurate, perhaps even significantly under-reporting the error coverage of their technique. In general, ACE analysis will have difficulty providing accurate reliability estimates for any partial coverage error mitigation technique whose placement of error coverage is commonly affected by the presence of soft errors.

#### 4. ACE ANALYSIS REFINEMENTS

There are two key contributors to conservatism in ACE analysis: limited context in ACE analysis and Y-Bits. We will analyze these two contributors in the subsections that follow.

##### 4.1 Refined ACE analysis

While determining the ACE criteria in Table 1, it was readily apparent that some of the labeling was very conservative. For example, the program counter field comprises 20 of the 88 bits in each scheduler entry. The field is only utilized when a control instruction is present, but given that we only know if the scheduler entry is occupied and whether any occupying instruction is ACE, we are forced to assume that all bits in the field are ACE whenever an ACE instruction occupies the scheduler entry.

By adding a small amount of complexity to the ACE analysis algorithm, we can keep track of whether or not a given ACE instruction is also a control instruction. This small amount of complexity allows further derating of the program counter field, increasing the accuracy of the analysis.

We used this refinement process on other fields in the scheduler entries as well. In addition to the program counter field, the source and destination register file pointers were derated whenever they were not used (21 bits in all — 3 of the 24 bits are actually enable bits). Furthermore, 24 of the 32 functional unit control, branch unit control, and miscellaneous control bits were also further derated. In all, we refined the ACE analysis for 65 of the 88 bits in each scheduler entry.

The fault masking level of IVM’s scheduler and the error coverage of ReStore obtained from using the refinements are presented in Figures 7 and 8. These figures present the results in the same manner as Figures 5 and 6 from the previous section. The old data is included in the figures as a reference, and the new data is labeled *ACE-refine*.

On average, the refined ACE analyses estimate a fault masking level of 76% for the scheduler in the baseline processor (up from 69%). This represents a 33% reduction in the gap between the original ACE estimate and the fault injection analysis. ReStore’s estimated fault coverage increases slightly from 38% to 46%, yielding an overall fault masking level of 87% for the ReStore processor’s scheduler. (up significantly from 81%). Using the same math as before, the difference between the ACE analysis and fault injection estimate of the reliability of the ReStore scheduler is now 26x (down from 38x).

Figure 9 plots the analysis difference for each type of scheduler entry state. As usual, the y-axis plots the level of fault masking. Table 1 lists the abbreviations for the bars in the plot. Note that while each of the bars spans 100%, the categories are not uniformly sized in the scheduler. Thus, the different categories have varying levels of impact on the

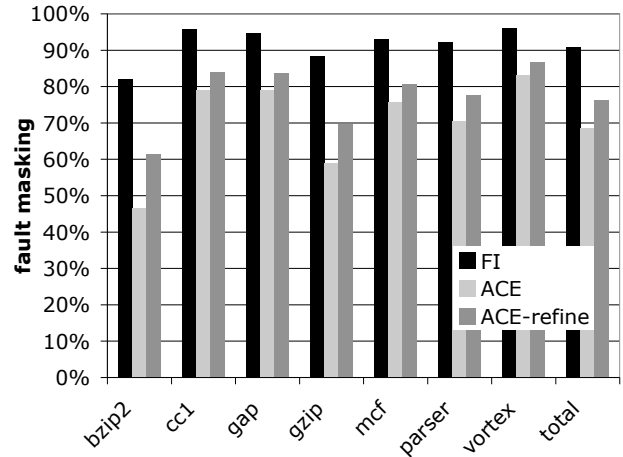


Figure 7: Refined ACE analysis of fault masking in the baseline processor’s scheduler.

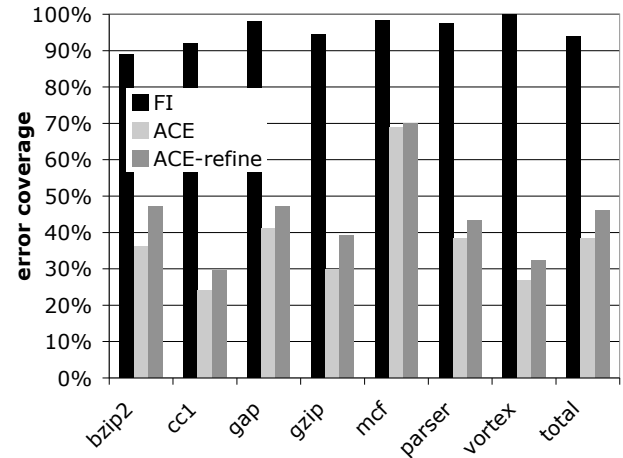


Figure 8: Refined ACE analysis of error coverage in the ReStore processor’s scheduler.

aggregate result. In particular, the program counter, functional unit control, and register pointer fields account for 58 of the 88 bits in each entry.

Derating the program counter and branch unit control fields proved to be very productive. This is because in the previous analysis, many non-control instructions were forcing the fields to be labeled ACE while they were unACE. Derating the functional unit control and miscellaneous control fields had little effect because they derated fields that were unused for relatively rare instructions like multiplies and conditional moves. Finally, derating the register file pointers also had marginal effect.

The valid field has a 27% vulnerability factor despite its “always ACE” classification. It turns out that due to the nature of the scheduling algorithm implemented in IVM, valid bits are often unACE when they are clear (invalid).

While a significant portion of the scheduler entry was derated further in this revised ACE analysis study, the reliability estimates were still fairly conservative. The refined ACE analysis results differ from our benchmark fault injection

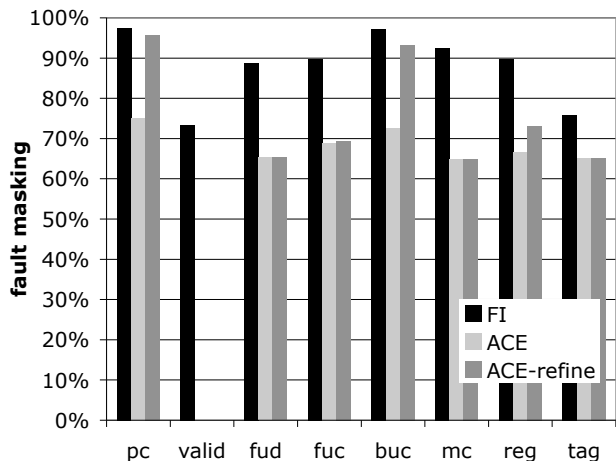


Figure 9: Estimated fault masking for each type of state in the baseline scheduler.

analysis result by factors of 2.6x (for the baseline processor) and 26x (for the ReStore processor).

The take-away is that a “simple” ACE analysis of the IVM scheduler revealed a 3-4x discrepancy in analyzing inherent fault masking. Simple refinements were able to lower the discrepancy to the 2-3x range. Architects utilizing the ACE analysis methodology to estimate the reliability of their designs can use these figures to help interpret their own experimental results.

Theoretically (ignoring human labor and computational cost), one can continually refine ACE analysis until there is no more benefit to be obtained. At this point, some conservatism still remains — we term the cause of this conservatism *Y-Bits*, which we will explore in the next section. However, before this theoretical limit for ACE analysis is reached, a practical threshold is encountered. ACE analysis quickly becomes intractable with increasing detail due to state explosion. An example of this was given in Section 3.5 when we discussed the difficulty of utilizing ACE analysis to evaluate architectures like ReStore. It is possible but perhaps not practical to track bit corruptions as they propagate to soft error symptoms in a single-pass simulation. We believe the practical limit is not far off from our refined analysis, at least in terms of fault coverage.

## 4.2 Limitations of ACE analysis: Y-Bits and missing bits

Y-Bits are dynamic bits of state that have the power to dramatically alter the course of execution in the processor, yet **do not** ultimately affect architecturally correct execution. The existence of Y-Bits was previously noted in [1] in association with analysis of a tag SRAM array. The effects that these bits have cannot be analyzed in a framework where analysis of all bits in the machine are conducted in a single simulation pass. ACE analysis assumes that Y-Bits are ACE, introducing another source of conservatism. Note that Y-Bits are not static bits of state. Instead they are dynamic instances of static bits of state.

An example of a Y-Bit is a bit of state in a specific clock cycle that, when corrupted, induces a Y-Branch [11]. A

Y-Branch is defined as a conditional branch that executes legally but incorrectly (i.e. it causes the fall-through path to be followed instead of the taken path, or vice-versa), yet does not adversely affect the outcome of the program. A Y-Bit that induces a Y-Branch causes the flow of the processor to deviate significantly, since a different sequence of instructions is fetched and retired. Thus, traditional ACE analysis that utilizes a single pass simulation cannot fully analyze the effect of corrupting such a Y-Bit.

It may be possible to meld ACE analysis with fault injection to analyze potential Y-Bits, but this topic is beyond the scope of this paper. We note that general fault injection into performance models is difficult since performance models typically lack mechanisms to accurately capture the effects of injected faults.

Processor state that was excluded from ACE analysis is also a source of conservatism. For example, in our instruction scheduler case study, we excluded the state related to several state machines from our analysis. This state was excluded because bit corruptions can have wide reaching effects and thus their effects are difficult to evaluate in the single-pass ACE analysis framework. In this sense, the state excluded from ACE analysis is a source of Y-Bits.

Looking through the 40,000 bits of state in IVM, we found that about 3.5% of the bits would be excluded from typical ACE analysis studies. Most of this state is in the form of latches as opposed to SRAMs. Out of approximately 13,000 latches in the processor, about 8% would be excluded from typical ACE analysis studies. Examples include valid bits, write enable bits, state machine state, and the program counter register.

A fault injection campaign restricted to the 3.5% of state excluded from ACE analysis revealed a fault masking rate of 75%, meaning that only 25% of injected faults propagated to DUE or SDC. Thus, for this set of processor state, Y-Bits contribute significantly to ACE analysis conservatism (ACE analysis would assume that these bits are 100% vulnerability while the fault injection campaign returned 25%). While the 75% masking rate is significantly lower than that of the remainder of the processor (about 3x more faults slip through), the overall impact on the reliability of the processor is small owing to the small subset of processor state.

## 5. RELATED WORK

There are three main bodies of related works: ACE analysis methodology conception, statistical fault injection methodology, and research that utilized ACE analysis.

Mukherjee et al. [6] introduced ACE analysis. Biswas et al. [1] extended the original ACE analysis framework to enable analysis of address based processor structures.

Wang et al. [12, 13] outlined statistical fault injection methodologies to evaluate the fault masking levels of a baseline processor architecture and an architecture augmented with ReStore. Kim and Somani [4] and Czeck and Siewiorek [2] utilize statistical fault injection methodologies to investigate the fault tolerance of microprocessors as well.

In this work, we identify and analyze the conservatism introduced in ACE analysis. We accomplish this task by utilizing a detailed fault injection analysis methodology as a benchmark to compare against ACE analysis.

There have been several bodies of research that utilized

ACE analysis methodologies. Mukherjee et al. [5] investigated whether cache scrubbing is necessary given today’s raw FIT rates and typical cache structures. Weaver et al. [15] utilize ACE analysis to estimate reliability improvement as a result of implementing a microarchitectural mechanism to avoid soft error events when convenient. Gomaa and Vijaykumar [3] utilize ACE analysis to estimate reliability improvement as a result of implementing an architecture that selectively performs transient fault detection whenever economical. Finally, Reis et al. [9] utilize ACE analysis in their investigation of a hybrid hardware/software transient fault detection architecture.

Our work in analyzing the conservatism introduced in ACE analysis enables a more complete interpretation of the research results reported by these ACE analysis based studies. For example, if a study utilizes a relatively crude ACE analysis methodology to estimate fault coverage, the reader can use the insights presented in this work to calibrate the reported results.

## 6. CONCLUSIONS

The ACE analysis methodology acts to identify bits of state that are known to not be critical for architecturally correct execution (ACE). Because of this fundamental principle, ACE analysis inherently delivers a conservative estimate of the fault tolerance of a processor architecture. A conservative estimate provides a lower bound on the reliability of a processor, which is useful in demonstrating that a design will meet reliability goals. This is particularly true when bounds are tight or when goals are lax or easily met.

In this paper, we quantify the degree of conservatism in ACE analysis by comparing ACE analysis to a rigorous fault-injection study. We observe that an ACE analysis of an instruction scheduler overestimates soft error vulnerability by about 3.5x. Refinements to the ACE analysis methodology reduce this figure to about 2.6x at the cost of additional simulation complexity. The discrepancy between ACE analysis and fault injection primarily stem from lack of detail (and difficulty in efficiently adding detail) and single-pass simulation methodology.

We also utilized ACE analysis to estimate the error coverage of ReStore, a partial coverage soft error mitigation technique. We found that ACE analysis under-reported error coverage by a significant margin, owing to a combination of conservatism in evaluating the baseline processor and ReStore’s symptom-based error detection scheme. Other error mitigation techniques that share ReStore’s attributes will observe the same consequences under ACE analysis.

Nevertheless, ACE analysis remains an indispensable tool in providing an early estimate and lower bound of processor reliability. The results of this study help bridge the gap between reliability estimates provided by ACE analysis and the actual fault tolerance of the design.

## 7. ACKNOWLEDGMENTS

The authors wish to thank Babak Falsafi and the anonymous referees for providing feedback during various stages of this work. This work was supported by the C2S2 Marco center, NSF grant EIA-0224453, and equipment donation from AMD.

## 8. REFERENCES

- [1] A. Biswas, P. Racunas, R. Cheveresan, J. Emer, S. S. Mukherjee, and R. Rangan. Computing the Architectural Vulnerability Factor for address-based structures. In *ISCA-33*, pages 532–543, June 2005.
- [2] E. W. Czeck and D. Siewiorek. Effects of transient gate-level faults on program behavior. In *Proceedings of the 1990 International Symposium on Fault-Tolerant Computing*, pages 236–243, June 1990.
- [3] M. A. Gomaa and T. N. Vijaykumar. Opportunistic transient-fault detection. In *ISCA-33*, pages 172–183, June 2005.
- [4] S. Kim and A. K. Somani. Soft error sensitivity characterization for microprocessor dependability enhancement strategy. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 416–425, Sept. 2002.
- [5] S. S. Mukherjee, J. Emer, T. Fossum, and S. K. Reinhardt. Cache scrubbing in microprocessors: Myth or necessity? In *10th IEEE Pacific Rim International Symposium on Dependable Computing*, pages 37–42, Mar. 2004.
- [6] 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 *MICRO-36*, pages 29–40, Dec. 2003.
- [7] P. Racunas, K. Constantinides, S. Manne, and S. S. Mukherjee. Perturbation-based fault screening. In *Proceedings of the 13th International Symposium on High-Performance Computer Architecture*, Feb. 2007.
- [8] S. K. Reinhardt and S. S. Mukherjee. Transient fault detection via simultaneous multithreading. In *ISCA-27*, June 2000.
- [9] 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 *ISCA-33*, pages 148–159, June 2005.
- [10] E. Rotenberg. AR-SMT: A microarchitectural approach to fault tolerance in microprocessors. In *FTCS*, June 1999.
- [11] N. Wang, M. Fertig, and S. Patel. Y-branches: When you come to a fork in the road, take it. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pages 56–66, 2003.
- [12] N. J. Wang and S. J. Patel. Restore: Symptom based soft error detection in microprocessors. In *DSN-2005*, June 2005.
- [13] 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 *DSN-2004*, June 2004.
- [14] C. Weaver and T. Austin. A fault tolerant approach to microprocessor design. In *ISCA-29*, May 2002.
- [15] C. Weaver, J. Emer, S. S. Mukherjee, and S. K. Reinhardt. Techniques to reduce the soft error rate of a high-performance microprocessor. In *ISCA-31*, June 2004.