Appears in Proceedings of the 15th International Symposium on High-Performance Computer Architecture, February, 2009.

Accurate Microarchitecture-Level Fault Modeling for Studying Hardware Faults *

Man-Lap Li, Pradeep Ramachandran, Ulya R. Karpuzcu, Siva Kumar Sastry Hari, Sarita V. Adve
Department of Computer Science
University of Illinois at Urbana-Champaign
swat@cs.uiuc.edu

Abstract

Decreasing hardware reliability is expected to impede the exploitation of increasing integration projected by Moore's Law. There is much ongoing research on efficient fault tolerance mechanisms across all levels of the system stack, from the device level to the system level. High-level fault tolerance solutions, such as at the microarchitecture and system levels, are commonly evaluated using statistical fault injections with microarchitecture-level fault models. Since hardware faults actually manifest at a much lower level, it is unclear if such high level fault models are acceptably accurate. On the other hand, lower level models, such as at the gate level, may be more accurate, but their increased simulation times make it hard to track the system-level propagation of faults. Thus, an evaluation of high-level reliability solutions entails the classical tradeoff between speed and accuracy. This paper seeks to quantify and alleviate this tradeoff.

We make the following contributions: (1) We introduce SWAT-Sim*, a novel fault injection infrastructure that uses hierarchical simulation to study the system-level manifestations of permanent (and transient) gate-level faults. For our experiments, SWAT-Sim incurs a small average performance overhead of under 3x, for the components we simulate, when compared to pure microarchitectural simulations. (2) We study system-level manifestations of faults injected under different microarchitecture-level and gate-level fault models and identify the reasons for the inability of microarchitecture-level faults to model gate-level faults in general. (3) Based on our analysis, we derive two probabilistic microarchitecture-level fault models to mimic gate-level stuck-at and delay faults. Our results show that these models are, in general, inaccurate as they do not capture the complex manifestation of gate-level faults. The inaccuracies in existing models and the lack of more accurate microarchitecture-level models motivate using infrastructures similar to SWAT-Sim to faithfully model the microarchitecture-level effects of gate-level faults.*

*This work is supported in part by the Gigascale Systems Research Center (funded under FCRP, an SRC program), the National Science Foundation under Grants CCF 05-41383, CCF 08-11693, and CNS 07-20743, an OpenSPARC Center of Excellence at the University of Illinois at Urbana-Champaign supported by Sun Microsystems, and an equipment donation from AMD.

1 Introduction

While technology scaling facilitates extended system integration, the scaled transistors are increasingly prone to failures for reasons such as infant mortality, wear-out, variation, etc., making them less reliable. The hardware reliability problem has, in the past, concerned only high-end niche systems where solutions that involve heavy amounts of redundancy in terms of space, time, or information are acceptable. In the future, however, the reliability problem is expected to pervade even the mainstream computing market where traditional solutions are too expensive to be applied.

To counter this reliability threat, researchers have proposed solutions at all levels of design, from the system level all the way down to the circuit and the device level. Examples include software-level symptom-based detection techniques such as SWAT that capture how hardware faults manifest to the system level [12, 22], end-to-end error detection and correction [23], microarchitecture-level (μarch-level) redundancy [27], and circuit-level BIST techniques [6].

To evaluate the efficacy of these solutions, it is essential to capture the expected behavior of the fault at the level at which the solution is implemented. For example, the manifestation of a gate-level floating-point (FP) unit fault needs to be accurately captured at the microarchitecture level to evaluate the efficacy of a proposed microarchitecture-level floating-point unit checker. This paper concerns accurate models of hardware faults at the microarchitecture level to evaluate fault-tolerant solutions at the microarchitecture and higher levels.

Recently, several μarch-level solutions that tolerate hardware failures have been proposed [2, 8, 12, 14, 25, 30]. The primary evaluation mode for these proposals has been through statistical fault injections in simulations either at the gate level [8, 14, 25] or the microarchitectural state elements (e.g., output latch of an ALU) [2, 12, 30]. While gate-level fault injections can accurately capture lower level faults, the long simulation time of these schemes prevents detailed evaluation of the propagation of gate-level faults through the hardware and into the software. On the other hand, the μarch-level injections are fast and allow observing faults propagated to the software level. However, while latch-level injections may be appropriate for array elements within the processor, it is unclear whether modeling faults in combinational logic at the latch level (e.g., injecting a fault

at the output latch of the FP unit to represent a fault in the logic), is accurate. While alternative FPGA-based emulations [11, 17, 21] offer higher speed and model gate-level faults with high fidelity, the limited observability and controllability gives less flexibility than software simulations. Hence, this paper focuses on software simulation methods.

The lack of speed in the gate-level fault simulation paradigm and the possible lack of fault modeling fidelity in μarch-level fault simulation prompt searching for a solution that can achieve the best of both worlds. To address this classic tradeoff between speed and accuracy, past work has applied the paradigm of hierarchical simulation, where different parts of the system are simulated at different abstraction levels so that required details are modeled only in the parts of interest, thus incurring reasonable performance overheads [1, 5, 7, 10, 15, 18].

In the context of fault tolerance, hierarchical simulations have been used to study transient faults in the processor by using a hierarchy of RTL and lower-level simulators [7, 15]. Since these simulators were used to study transients, they invoke the lower-level simulator just once to capture the effect of the fault, following which simulation happens only in the higher level. Other work has used hierarchical simulations to generate fault dictionaries that capture the manifestations from the lower level "off-line" and use them to propagate fault effects during high-level simulations [10]. This idea of fault dictionaries has also been used to study gate-level stuck-at faults in small structures, such as an adder [5]. However, fault dictionaries are specific to the fault model for which they are generated and cannot be used to simulate arbitrary fault models (the dictionary will have to be generated off-line for every such fault model); timing faults particularly present a challenge. Further, for faults in arbitrarily large structures, the growing sizes of inputs and faults make the dictionaries intractable, making them hard to use.

Our focus here is on the increasingly important permanent and intermittent faults [4, 31] and solutions for modeling them at the microarchitecture level or higher. In particular, successful solutions must address the following three critical aspects of fault simulation that prior work does not address in unison.

1. Simulation must be fast enough to capture how software would be affected by hardware faults.

2. Unlike transients, where the fault effect can be captured *once* and propagated to the higher abstraction level, permanent and intermittent faults have the characteristic that one activation of a fault could corrupt the software execution, which influences future activations of the same fault. This feedback mechanism between the hardware fault and the software must be faithfully simulated.

3. The simulator must be flexible enough to model different types of faults.

1.1 SWAT-Sim

To meet the stated criteria, we propose a novel fault injection infrastructure, *SWAT-Sim*, that couples a microarchitecture-level simulator with a gate-level simulator and has the following properties.

1. To achieve speed close to a microarchitectural simulator and minimize overhead, SWAT-Sim only simulates the component of interest (in our case, the faulty component) at gate-level accuracy and invokes a gate-level simulation of the component *on-demand*.

2. To accurately capture the interaction between the hardware fault and the software, SWAT-Sim invokes the gate-level simulation repeatedly during runtime (interspersed with μarch-level simulations); thus, if the software activates the gate-level fault, it would be corrupted and affects future activations of the same fault.

3. To allow fault modeling flexibility, SWAT-Sim employs a gate-level timing simulator where different timing faults can be modeled by changing the delay information within the faulty module.

These design choices of SWAT-Sim allow studying of the impact of gate-level permanent faults on software at speeds comparable to μarch-level simulators. Further, since the fault simulation is performed while real-world software is executing, the effect of the fault is studied using functional vectors that represent realistic scenarios. SWAT-Sim thus has an advantage over other methods that use artificially generated test vectors (e.g., functional vectors collected from a fault-free execution) to study the fault effect, as test vectors may not be representative of real-world faulty behavior.

1.2 Contributions

- We present SWAT-Sim, a novel fault injection infrastructure for studying system-level effects of gate-level permanent faults. To the best of our knowledge, SWAT-Sim is the first simulator that facilitates detailed understanding of *permanent* fault propagation from the gate level, through the microarchitecture, to the full system level, with real-world applications running on an operating system. SWAT-Sim is both fast (compared to gate-level simulators) and accurate (compared to μarch-level simulators), with a small average overhead of 3x, for the components we simulate, over μarch-level simulators.

- With SWAT-Sim, we study the system-level manifestations of faults injected in a Decoder, an Arithmetic and Logic Unit (ALU), and an Address Generation Unit (AGEN) of a superscalar processor. We inject faults under μarch-level stuck-at, gate-level stuck-at, and gate-level delay fault models, and use the previously studied SWAT detection techniques to understand their system-level manifestation [12]. We show that, in

general, μarch-level stuck-at faults do not result in similar system-level fault manifestation as gate-level stuck-at or delay faults. We thus infer that more accurate models are needed to model gate-level faults at the μarch level.

- Based on an extensive analysis of the propagation of gate-level faults to the microarchitecture, we derive two probabilistic fault models, the *P-model* and the *PD-model*, for gate-level stuck-at and delay faults. Our analysis suggests that these models are, in general, inaccurate μarch-level models for gate-level faults because they fail to capture the complex manifestation of gate-level faults. However, we identify several reasons of the inaccuracies of the models that could aid deriving better μarch-level models in the future.

Overall, this paper makes a first attempt towards understanding the differences in system-level effects between the μarch-level stuck-at fault models and gate-level stuck-at and delay fault models. Our extensive analysis and modeling showed that it is highly complex to capture the several factors that should be used for deriving μarch models that accurately represent the behavior of gate-level faults. Therefore, until there are further breakthroughs in μarch-level fault models, we believe that gate-level simulations are necessary to capture the behavior of gate-level faults. Hence, fast simulation methods, such as the proposed hierarchical simulator, SWAT-Sim, are essential for studying system-level effects of hardware faults.

2 The SWAT-Sim Infrastructure

SWAT-Sim is fundamentally a μarch-level simulator that only simulates the faulty μarch-level blocks, such as a faulty ALU or decoder, at the gate level. This greatly minimizes the gate-level simulation overhead.

2.1 Interfacing the Simulators

SWAT-Sim couples a full-system μarch-level and a gate-level simulator. A gate-level Verilog module of the faulty unit is simulated only when the unit is utilized by the μarch-level simulator. The inputs to the μarch-level unit are passed as stimuli to the gate-level simulator. When the gate-level simulation completes, the results are passed back to the μarch-level simulator, which then continues execution.

This communication between the two simulators is achieved using UNIX named pipes. In the μarch-level simulation, each time an instruction utilizing the faulty unit is encountered, the stimuli needed by the gate-level module are written to a dedicated stimuli pipe. After the gate-level simulation completes, the computed data is written to a dedicated response pipe from where the μarch-level simulator can read the response.

While the μarch-level simulator can access the named pipes like files, the gate-level simulator is enhanced with two system tasks, implemented using the Verilog Procedural Interface (VPI) [9], that handle accesses to/from the pipes: One collects signals from the stimuli pipe and the other writes the results to the response pipe. The stimuli and response (arguments of the two tasks) are tailored to the μarch-level structures under fault injection.

Figure 1 compares how a single fault in a μarch-level structure X is simulated in a purely μarch-level simulator (Figure 1(a)) and in SWAT-Sim (Figure 1(b)).

In Figure 1(a), a single fault in X is modeled as a single-bit corruption at the output latch of X because the μarch-level simulator lacks the gate-level details of X.

On the other hand, at the gate-level, a single fault in X is modeled as a fault in a specific gate or net. Figure 1(b) shows the steps of how the SWAT-Sim hierarchical simulator simulates the effect of this fault. (1) An instruction in the μarch-level simulator uses X. SWAT-Sim collects the relevant input vectors and sends them to the stimuli pipe. (2) The Verilog system task reads from the input pipe and sends the stimuli to the gate-level simulator. (3) The gate-level simulator feeds the stimuli to the faulty module and obtains the output after gate-level simulation. (4) The Verilog system task transfers the result from the gate-level simulator to the response pipe. (5) The μarch-level simulator reads the result from the response pipe and continues simulation. In particular, the figure shows the effect of a single gate-level fault propagating into a multiple-bit corruption at the output latch. In contrast, the fault injected in pure μarch-level simulation only results in a single-bit corruption (Figure 1(a)).

2.2 Different μarch-level Structures

Given the wide variety of structures within a modern processor and the differences in the abstraction levels between a typical μarch-level simulator and its corresponding gate-level counterpart, several factors should be considered when performing such hierarchical simulations.

Simulating sequential logic: Simulating combinational logic with single- or multi-cycle latency in SWAT-Sim is straightforward. As long as the outputs are read after the stipulated latency, the outputs are guaranteed to be correct for each invocation. Sequential logic, however, requires state to be maintained across invocations. In SWAT-Sim, since the gate-level simulator is invoked (and thus clocked) only when the unit is utilized, state is maintained across multiple invocations, resulting in accurate simulation of sequential circuits.

Handling gate-level signals that are not modeled at the μarch level: In some cases, due to abstract modeling in the μarch simulators, not all signals modeled at the gate-level appear at the μarch level. If the faulty component contains such signals, the μarch-level simulator can be enhanced with those signals to help propagate faults in these paths, improving its accuracy. Even in the absence of these enhancements, SWAT-Sim would present a more accurate fault model than existing μarch-level fault models.

Simulating large μarch-level components that may result in large overheads: Since the primary aim of SWAT-Sim is being able to study the propagation of gate-level faults to the system level, simulations must be carried out at reasonable speeds. The components we study in the paper present

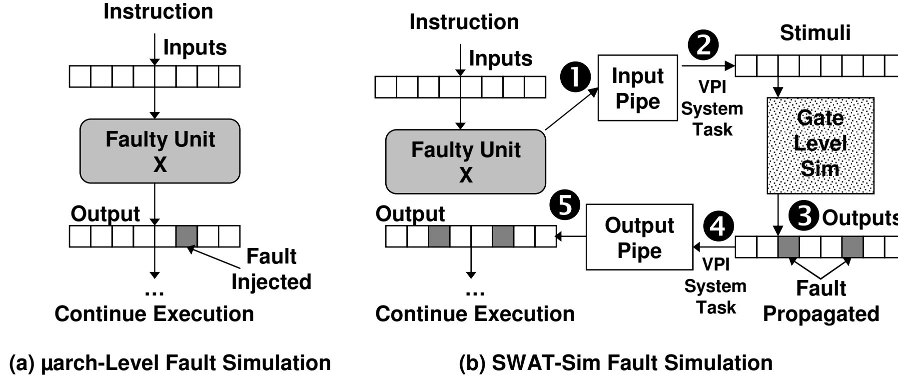


Figure 1. Comparison of how a faulty μarch-level unit X is simulated by (a) a pure μarch-level simulator and (b) by SWAT-Sim.

overheads in simulation time of under 3x (discussed in Section 4.1), when compared to pure μarch-level simulations. However, if the overhead becomes exorbitant because the faulty module is too large, the module can be further partitioned so that only the faulty submodule is simulated at the gate level while the rest is simulated at the higher level. For example, [15] uses such an approach in a lower-level hierarchical simulator.

Overall, by effectively coupling the gate-level and μarch-level simulators, SWAT-Sim is capable of simulating gate-level faults in different μarch-level components, making it a useful tool for full-system fault propagation studies with gate-level accuracy.

3 Experimental Methodology

3.1 Simulation Infrastructure

Since permanent faults are persistent and can propagate through the μarch-level to affect the OS and application state, SWAT-Sim requires a full-system, a μarch-level, and a gate-level timing simulator. Any set of such simulators may be interfaced for the purposes of fault propagation.

In our implementation, SWAT-Sim consists of three components – the Virtutech Simics full-system functional simulator [29], the Wisconsin GEMS processor and memory μarch-level timing models [13], and the Cadence NC-Verilog gate-level simulator. We interfaced the Cadence NC-Verilog simulator with GEMS using system calls implemented in VPI as described in Section 2.1

For the gate-level modules, we obtained the RTL designs of the arithmetic and logic unit (ALU) and the address generation unit (AGEN) from the OpenSPARC T1 architecture [28] and built an RTL model of the SPARC V9 decoder based on the decoder in GEMS. The Decoder module decodes one 32-bit instruction word per cycle and generates the signals modeled by our μarch-level simulator. The ALU module is capable of executing arithmetic (add, sub), logical (and, or, not, xor, and mov), and shift (shift-left and shift-right) instructions. The AGEN module computes the effective virtual address given the operand values of the memory (load/store) instruction. Using Synopsys Design Compiler, we synthesized these modules at 1GHz with the UMC 0.13μm standard cell library. Further, this synthesis tool also

Base Processor Parameters	
Fetch/Decode/ Execute/Retire rate	4 per cycle
Functional Units	2 Int add/mul, 1 Int div, 2 Load, 2 Store, 1 Branch, 2 FP add, 1 FP mul, 1 FP div/sqrt
Integer Unit latencies	1 add, 4 mul, 24 divide
FP Unit latencies	4 default, 7 mul, 12 divide
Reorder Buffer size	128
Register File size	256 integer, 256 FP
Unified Load-Store Queue	64 entries
Base Memory Hierarchy Parameters	
Data L1/Instruction L1	16KB each
L1 hit latency	1 cycle
L2 (Unified)	1MB
L2 hit/miss latency	6/80 cycles

Table 1. Parameters of the simulated processor.

generates the SDF (Standard Delay Format) file that contains the delay information of each gate and wire within the synthesized gate-level module. The Cadence NC-Verilog simulator then performs gate-level timing simulations with information provided in this file. For delay faults (described in Section 3.2), we modify the post-synthesis SDF file to incorporate added delays.

This simulation setup allows us to inject permanent faults under different fault models into the ALU, the AGEN, and the Decoder, and to observe their impact on real workloads (6 SpecInt2000 and 4 SpecFP2000) running on the Sun Solaris-9 operating system. Both the application and the OS run on a simulated 4-wide out-of-order processor (Table 1) supporting the SPARC V9 ISA.

3.2 Fault Models

In our experiments, we injected faults according to the following fault models to study differences in system-level effects among faults injected at the μarch level and the gate level. In all cases, we inject single bit (or single wire) faults.
Gate-level stuck-at fault model: The gate-level stuck-at fault model is a standard fault model applied in manufacturing testing. We inject both stuck-at-0 and stuck-at-1 faults in randomly chosen wires in the circuit.
Gate-level timing fault model: It has been shown that aging-related faults result in timing errors in the faulty gate,

with increasing delay as the aging worsens [3]. Ideally, we would like to model this effect using transition fault models and path delay faults, with different amount of delays. Here, we experiment with two delay fault models: (1) We inject a one-clock-cycle delay into the faulty gate such that timing violations occur along all paths containing the gate when a transition occurs. (2) The faulty gate is injected with a half-clock-cycle delay, potentially causing a subset of the gate's output cone to violate timing.

Microarchitecture-level stuck-at fault model: Due to the absence of more accurate fault models, stuck-at faults at the input/output latch of a faulty μarch-level unit have been used to estimate the effect of gate-level faults (both stuck-at and timing-related faults). We adopt this fault model, injecting both stuck-at-0 and stuck-at-1 faults in the input of the Decoder and the output latch of the ALU or AGEN.

3.3 Studying System-Level Effects

A key objective of this study is to understand the differences, if any, in system-level manifestations of μarch-level and gate-level faults within μarch-level structures. For this purpose, we use the SWAT symptom-based detection scheme because these detectors essentially capture how hardware faults manifest into the system level and software [12].

We inject faults using the fault models described in Section 3.2 and rely on SWAT-Sim's full-system μarch-level simulator to propagate the fault effect to the software for the SWAT detectors to detect. Specifically, we use the following SWAT detectors – (1) FatalTraps, such as memory address misalignment, illegal instruction, etc., denoting an abnormal software execution, (2) Hangs of the application and the OS, identified using a hardware hang detector, and (3) HighOS, representing abnormal executions that have excessive contiguous OS instructions (30,000 contiguous instructions for our experiments) [12]. Following the methodology used in [12], we consider faults detected within 10 million instructions (after corruption of some architectural state) to be recoverable (e.g., using pure hardware [19, 26], or hybrid hardware/software recovery schemes). Therefore, for each fault injection run, SWAT-Sim performs a detailed timing simulation (both μarch and gate-level) for 10 million instructions after the first architectural state corruption.[1] If there is no architectural state corruption for 10 million instructions after the fault injection, the fault is assumed to be masked and the simulation is terminated.

Thus, at the end of the above 10 million instruction window, a fault results in one of the following outcomes: (1) *μarch-Mask*: the μarch-level state (output latches of ALU, AGEN, and Decoder) is never affected. (2) *Arch-Mask*: the architectural state is not corrupted. (3) *Detected*: a detection occurs. (4) *Unknown*: the fault is neither detected nor masked at the μarch and architecture level.

The faulty cases that result in *unknown* are then simulated in functional mode and can have one of three outcomes: (1) *Application-level masking*: even though the fault corrupts the architectural state, the application output remains correct. (2) *Detected>10M*: the fault is detected later in the execution, but is deemed not recoverable. (3) *Silent data corruption (SDC)*: the fault remains undetected and corrupts the application output.

Given the injection outcomes, we study the differences between the various fault models using two metrics, coverage and detection latency, as follows.

Coverage: We define coverage as the percentage of unmasked faults that are detected within 10 million instructions and calculated as $\frac{Total\ Detected}{Injected\ -\ Masked} \times 100$, where μarch-Mask, Arch-Mask, and App-Mask constitute masked injections.

Detection Latency: The latency of detection determines the recoverability of the fault. Faults with shorter detection latencies can be fully recovered using hardware techniques (e.g., [19, 26]) with little hardware buffering to handle input/output commit problems. On the other hand, while the memory state corrupted by faults with longer detection latencies can be recovered using hardware techniques, the support for handling the input/output commit problems would be more complex and may require software involvement. We measure the latency of detection from the instruction at which the architectural state (of either the application or the OS) is corrupted until the detection.

3.4 Parameters of the Fault Injection

Our fault injection campaign consists of several runs for each of our 10 applications, with 1 fault injected per run. For each combination of fault model (Section 3.2), faulty structure, and application, we inject a fault in one of 4 different randomly chosen points in the application and one of 50 different points in the faulty unit. For the gate-level stuck-at and delay fault models, the 50 points in a structure are chosen from the 1853, 2641, and 757 wires of the synthesized gate-level representation of the Decoder, ALU, and AGEN respectively. For the μarch-level faults, these points are randomly chosen from the 32 bits of the input latch of the Decoder and from the 64 bits of the output latches of the ALU and AGEN. Further, since there are multiple decoders, ALUs and AGEN units in our superscalar processor, one of them is chosen randomly for each injection. We also ensure that the samples are chosen so that gate-level stuck-at and delay faults are injected in the same set of wires to facilitate a fair comparison among the gate-level faults.

This gives us a total of 2000 simulations per fault model per structure ($4 \times 10 \times 50$). Each injection run whose fault is not masked is a Bernoulli trial for coverage (either detected or not). Further, since the injection experiments are independent of each other, this gives us a low maximum error of 1.1% for the reported coverage numbers, at a 95% confidence interval.

3.5 Limitations of the Evaluation

Here, we list some of the assumptions and limitations of our evaluation.

[1] Architectural state corruptions are determined by continuously comparing against a golden fault-free run (see [12]).

- SWAT-Sim assumes that a Verilog description of the module of interest is readily available for interfacing. This is true for the large fraction of the processor that is typically re-used from older tape-outs. However, for modules that are yet to be developed, neither SWAT-Sim nor pure gate-level simulators can be used to perform fault injection experiments. As these models start to become available, SWAT-Sim can be incrementally interfaced with them.

- Using SWAT-Sim, we study the propagation of gate-level faults in only three microarchitecture units (Decoder, ALU, and AGEN) as we could not find other Verilog modules close enough to the SPARC architecture modeled by the μarch-level simulator (we used the in-order UltraSPARC T1 as our Verilog source and the out-of-order GEMS as our μarch-level source).

- The timing information generated in the SDF file represents pre-layout timing, which does not reflect accurate post-layout timing for both gate delays and interconnect. By extracting this information using a place-and-route tool, the accuracy of our timing simulations, and thus our results, can be further improved.

- Although prior work has suggested other statistical delay models for timing faults (e.g., based on threshold voltage and temperature [16, 24]), we inject fixed and arbitrarily chosen delay that may or may not represent real-world failure modes. Integrating more accurate lower-level timing fault models in SWAT-Sim is a subject of our future work.

In spite of these assumptions and limitations, the results presented in this paper demonstrate the importance of using hierarchical simulators, such as SWAT-Sim, to accurately model gate-level faults at the μarch level.

4 Results

The hierarchical nature of SWAT-Sim allows us to achieve gate-level accuracy in fault modeling, at speeds comparable with μarch-level simulators. We first summarize SWAT-Sim's performance when compared to both the μarch-level simulation and pure gate-level simulation (Section 4.1). We then use the SWAT-Sim simulator to first evaluate the accuracy of the previously used μarch-level stuck-at fault models for representing gate-level faults (Section 4.2). Subsequently, we extensively analyze the reasons for the differences in the manifestations of gate-level faults from μarch-level faults (Section 4.3). From this detailed analysis, we derive two candidate probabilistic μarch level fault models for modeling gate-level stuck-at and delay faults (Section 4.4).

4.1 Performance Overhead of SWAT-Sim

To understand the performance overhead incurred by SWAT-Sim when compared with pure μarch-level simulation, we profile a set of 40 fault-free runs for each structure and each fault model. We do not inject a fault in the desired faulty unit, but force the unit to be simulated at the gate level. To be conservative, we always use the most utilized unit for this purpose (e.g., ALU 0 for faulty ALU). For delay faults, we simulate the chosen unit with SDF timing annotation. Table 2 shows the maximum and average slowdowns of SWAT-Sim compared to pure μarch-level simulation, when simulating the ALU, the AGEN, and the Decoder across different fault models.

Unit	Fault Model	Max	Avg
ALU	Gate Stuck-At	2.20	1.56
	Gate Delay	2.65	1.93
AGEN	Gate Stuck-At	1.59	1.26
	Gate Delay	1.89	1.35
Decoder	Gate Stuck-At	2.91	2.12
	Gate Delay	5.10	2.91

Table 2. Slowdowns of SWAT-Sim when compared to pure μarch-level simulation.

Overall, the worst average-case slowdown of SWAT-Sim, compared to the μarch-level simulation, is under 3x, which is an acceptable overhead considering SWAT-Sim's ability to model gate-level faults. In particular, Table 2 shows that the Decoder incurs the most overhead, with average slowdowns of gate-level stuck-at and delay faults being 2.12x and 2.91x respectively. The average slowdowns of the ALU and the AGEN are under 2x. The maximum slowdowns observed for the ALU and the AGEN are under 2.7x and 2x, respectively while the overall maximum slowdown of 5.1x is measured for the Decoder. The Decoder incurs higher overhead than other units because it sits at the processor front-end and is more utilized than the ALU and the AGEN.

As expected, the delay fault simulations always incur higher overhead than the stuck-at fault simulations because simulating delay faults requires timing information which is more compute-intensive.

Since we do not have the corresponding gate-level model of the superscalar processor we simulate at the μarch level, we cannot directly determine the performance benefit of SWAT-Sim over pure gate-level simulation. Instead, we derive a rough conservative estimation of the performance benefit as follows. Assume (conservatively) that we need to simulate a fault in a circuit that contains 4 times the number of gates and is utilized twice as often as the Decoder. Assume that the full superscalar processor we wish to simulate has 25 million gates. Assuming SWAT-Sim's worst-case slowdown is linear to the utilization and the size of the gate-level module and the baseline μarch simulator simulates at the rate of 17k instr/sec (which is the measured average speed of our μarch-level simulator), it would take SWAT-Sim $10M\ instr \times \frac{4 \times 2 \times 5.1}{17k\ instr/sec} = 6.7\ hr$ to simulate 10 million instructions in the worst case. On the other hand, conservatively assuming the gate-level simulator simulates 25M gates-cycles/sec (more than 1300x the speed reported in [20]) and the execution has an IPC of 1, it would take $10M\ instr \times \frac{25M\ gates}{1\ instr/cycle \times 25M\ gates-cycles/sec} = 2778\ hr$ to simulate 10 million instructions. SWAT-Sim thus achieves

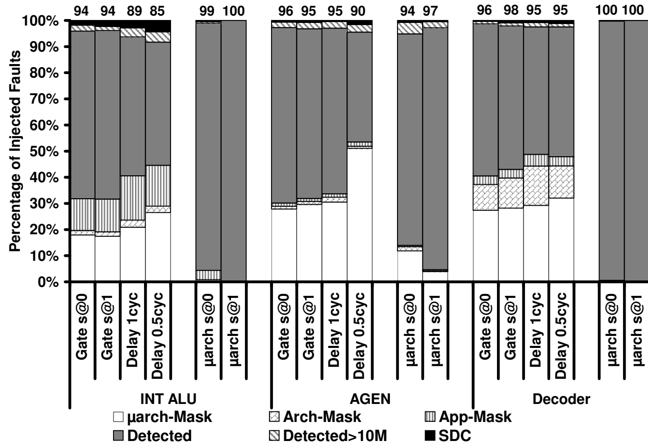


Figure 2. Efficacy of the SWAT fault detection scheme [12] under different fault models for the ALU, AGEN, and Decoder. Depending on the fault model and the structure, the μarch-level fault may or may not capture the system-level effects of gate-level faults accurately, as indicated by the differences in coverage.

a 417x speedup over traditional gate-level simulation.

4.2 Accuracy of Microarchitecture-Level Fault Models

We next investigate the accuracy of μarch-level fault models. If these fault models were accurate enough, then we can eliminate gate-level simulations entirely, thus eliminating the need for SWAT-Sim and its overhead.

4.2.1 Comparison of Coverage

Figure 2 compares the efficacy of the SWAT scheme in detecting different faults injected using different fault models into the ALU, the AGEN, and the Decoder. The bars represent the outcomes for the μarch-level stuck-at-1 (μarch s@1) and stuck-at-0 (μarch s@0) models, the gate-level stuck-at-1 and stuck-at-0 models (Gate s@1 and Gate s@0, respectively), and the gate-level 1-cycle-delay and 0.5-cycle-delay models (Delay 1cyc and Delay 0.5cyc, respectively). Each bar shows the fraction of fault injections that are microarchitecturally masked (*μarch-Mask*), architecturally masked (*Arch-Mask*), application-masked (*App-Mask*), detected within 10M instructions (*Detected*), detected beyond 10M instructions (i.e., not recoverable) (*Detected >10M*), and those that lead to silent data corruptions (*SDC*). The number on top of each bar represents the coverage.

Figure 2 shows that depending on the structure and the fault model, the μarch-level fault model may or may not accurately capture the effect of gate-level faults, as indicated by the coverage. For the AGEN, the coverage of μarch stuck-at faults is similar to that of the gate-level stuck-at and 1-cycle delay fault models (between 94% and 97%). However, the coverage of 0.5-cycle delay AGEN faults is noticeably lower (90%). For the Decoder and the ALU, the coverage for the μarch-level stuck-at faults is near perfect (99+%) while the coverage of the gate-level stuck-at faults (94% for the ALU and between 96% and 98% for the Decoder) and the Decoder delay faults (95%) is slightly more pessimistic. In contrast, the coverage of the ALU delay faults is significantly lower (89% and 85% for 1-cycle and 0.5-cycle delay faults, respectively).[2]

The following analyzes the faults that do not result in detection in more detail.

Masking: A large source of discrepancy among the different fault models lies in the masking rate (μarch-level, architectural, and application masking). The μarch-level stuck-at fault models have very little masking of all three kinds (on an average, 0.3% for the Decoder, 2% for the ALU, and under 9% for the AGEN), while the gate-level fault models show a much higher rate of masking (>30% for all structures, with 0.5-cycle delay faults in the AGEN having the highest masking rate of 54%).

The masking rates of μarch-level faults are low mainly because the faults are rarely μarch-masked when compared to gate-level faults. As μarch-level faults directly change the latch data, the only case where it does not result in a μarch corruption (i.e., is μarch-masked) is when the data does not activate the latch fault, e.g., correct data value of 0 masks a stuck-at-0 fault. At the gate level, there are two scenarios: (1) the fault at the gate is not activated, and (2) the fault is activated but does not propagate due to other signals in the circuit. Thus, the gate-level faults see much higher μarch masking rates. Further, the μarch-level faults are hardly masked at the application and architecture levels since the they tend to perturb the data more severely and cause symptoms more easily than the gate-level faults.

Interestingly, gate-level faults injected into the 3 structures exhibit different masking behaviors. All structures have high μarch-level masking. However, architectural masking is significant only for the Decoder (25% to 31%) and application masking is substantial only for the ALU (35% to 42%).

Decoder faults are more likely to be masked at the architecture level than other structures. For these cases, we observe that the faults affect a subset of instructions of types that are sparingly used and corrupt only wrong-path instructions. Thus, even though the gate-level faults become microarchitecturally visible, they are not activated again after the pipeline flush and thus the fault becomes architecturally masked. For the ALU and AGEN, however, we see relatively few faults that get activated only by speculative instructions.

On the other hand, a significant number of ALU faults are masked by the application. This is likely due to the activated faults being logically masked. For example, suppose instruction $r1 \leftarrow r2 + r3$ uses the faulty ALU and the fault causes $r1$ to change from 1 to 2. If $r1$ is only used for the branch instruction $beq\ r1, 0, L$, the fault effect is masked by the application. This type of masking is relatively rare in other structures. Since it is more likely for Decoder faults to affect the program control flow and for AGEN faults to change the addresses of memory accesses, these faults, once activated,

[2] We found the coverage with SWAT-Sim improves significantly (from 89% to 94% for 0.5-cycle delay faults in ALU) when the undetected cases are run for 50M instructions, showing that SWAT's detectors remain effective at this longer latency (which is still recoverable [19]).

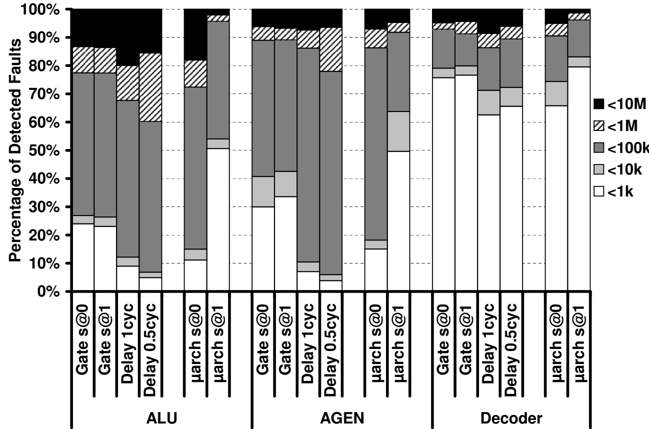


Figure 3. Latency of fault detection in terms of number of instructions executed from architectural state corruption to detection. The differences in the models impact recovery, which is primarily governed by these latencies.

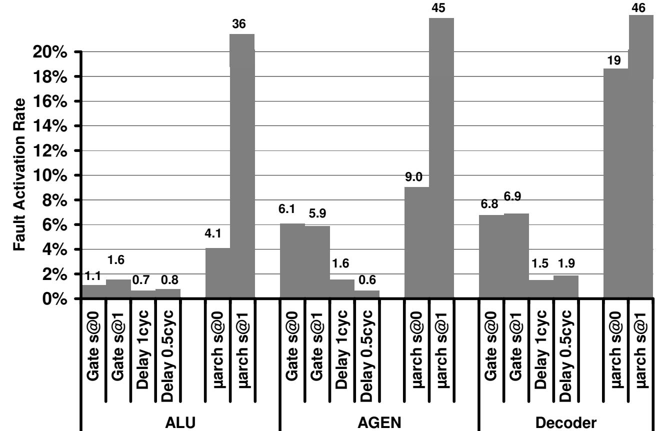


Figure 4. Mean fault activation rate for the different fault models as a percentage of the number of instructions.

usually lead to detectable symptoms (i.e., not masked).

SDC: Similar to the overall coverage, the SDC rates (percentage of total injections that result in SDC events) are dependent on the type of fault and the structure in which the fault is injected. While the SDC rate is higher for gate-level faults than μarch-level faults in the ALU (1.8%–4.4% vs. 0%–0.5%, respectively) and the Decoder (0.4%–1.2% vs. 0.1%–0.2%, respectively), the SDC rates of the AGEN faults are nearly identical (1.6% for 0.5-cycle delay faults and 0.5%–0.8% for others).

The SDC rates are high for the gate-level faults in the ALU because these faults are rarely activated and only perturb the data value slightly once activated. In contrast, the μarch-level stuck-at faults are easily activated and less likely to cause SDCs.

The above differences in manifestations are largely governed by how the fault at the gate level becomes visible to the microarchitecture (activation rate, which latch bits are corrupted, etc.), as analyzed further in Section 4.3.

4.2.2 Latency to Detection

We next discuss how the μarch- and gate-level fault models compare in terms of detection latency.

Figure 3 gives the total number of instructions executed after the architectural state is corrupted, until the fault is detected, for each unit under each fault model. The detected faults are binned into different stacks of the bar based on their detection latencies (from 1,000 to 10 million instructions).

As mentioned previously (Section 3.3), the latency to detection has direct bearing on the recoverability of the application and the system. Hardware buffering, required for hardware recovery, can buffer 1000s of instructions, but can be expanded through intelligent design to tolerate latencies of up to millions of instructions [19, 26].

From Figure 3, we see that the percentage of detected faults for which the software can be recovered using recovery techniques that can tolerate short latencies of under 10K instructions is different under different fault models for the three structures. While the μarch-level stuck-at-1 model shows that a larger fraction of faults are recoverable for the above latency than gate-level stuck-at faults, the recoverable faults for μarch-level stuck-at-0 faults is lower.

From these differences in system-level manifestations, we infer that μarch-level stuck-at faults do not, in general, accurately represent gate-level stuck-at or delay faults. This motivates either building more accurate μarch-level fault models, or in their absence, using the SWAT-Sim infrastructure to study the system-level effect of gate-level faults.

4.3 Differences Between Fault Models

Before we attempt to derive a more accurate μarch-level fault model than the existing ones, we investigate the fundamental reasons for the different behaviors of the μarch-level and gate-level fault models. In the following sections, we try to understand the differences by comparing the fault activation rates and the data corruption patterns at the microarchitectural state across different fault models.

4.3.1 Fault Activation Rates

The fault activation rate of a given faulty run is defined as the percentage of instructions that get corrupted by the injected fault among all instructions that utilize the faulty unit. We collect the activation rates for all faulty runs that do not result in μarch-masked, calculate the weighted arithmetic mean for each fault model, and present these numbers in Figure 4. Because the different runs execute different numbers of instructions, we weight the activation rate of each run by the total number of instructions executed by the faulty unit and calculate the weighted mean.

Figure 4 shows that the μarch-level stuck-at faults present a higher activation rate than faults injected at the gate-level. For the ALU, the μarch-level faults have a $>$4% activation rate, while the activation rates of gate-level faults are at most 1.6%. For the AGEN, the corresponding numbers are $>$9% and $<$7% respectively. The Decoder faults tend to have higher activation rates than faults in other structures because decoders are utilized more; the Decoder μarch-level faults have activation rates $>$19% while the rates of gate-level faults are $<$7%. The activation rate for gate-level faults

is lower because activating gate-level faults requires both excitation and propagation to the output latch, while the μarch-level fault is directly injected into the latch. Additionally, the μarch-level stuck-at-1 fault has a significantly higher activation rate than the other fault models (36%, 45%, and 47% for the ALU, the AGEN, and the Decoder respectively). This high rate is caused by the biases in data values towards zero.

Further, we notice a difference in the activation rates between the gate-level stuck-at and delay faults, with the delay fault models exhibiting lower rates of activation for all structures. Less than 2% of instructions activate the 1-cycle delay faults and 0.5-cycle delay faults in all 3 structures, with the lowest average activation rate being 0.6% for 0.5-cycle delay faults in the AGEN. The lower average activation rate can be explained with the different excitation conditions for the two models. A stuck-at-X fault is excited when the signal at the faulty net is $\overline{X}$. Thus, if the probability of having a logic 1 at the faulty net is *p*, the probability of exciting the stuck-at-0 fault at that wire is *p* and that of exciting the stuck-at-1 fault is *(1-p)*. A delay fault, on the other hand, is active only if there is a transition at the faulty wire and hence the excitation probability is *p(1-p)*, which is always smaller than that of the stuck-at faults. This lower probability of excitation generally results in a lower average activation rate for gate-level delay faults. Further, while an activated 1-cycle delay fault causes all paths from the faulty net to the output latch to miss timing, a 0.5-cycle delay fault usually results in fewer errors observed at the output as it can be the case that some paths from the faulty net to the output do not violate timing.

Although the higher activation rates (Figure 4) of μarch-level stuck-at faults result in higher coverage (Figure 2) for the ALU and Decoder, we do not find such a correlation for the AGEN. When comparing gate-level faults of the same structures, stuck-at faults have higher activation rates and result in slightly higher coverage than delay faults for the ALU and Decoder, but not for the AGEN. Nonetheless, higher activation rates do not necessarily drive the coverage up. Additionally, we find no direct correlation between activation rate and latency of detection. Thus, factors other than just activation rate need to be investigated if we are to succeed in deriving better μarch-level fault models. We next look at how activated faults manifest at the output latches (i.e., at the μarch-level).

4.3.2 Corruption Pattern at the Microarchitectural State

While an activated μarch-level fault corrupts only one bit in the microarchitectural state, an activated gate-level fault may corrupt multiple bits once it becomes visible in the microarchitectural state.

Table 3 shows the number of bits corrupted at the output latch (microarchitectural state) for different fault models for a fault in the ALU, the AGEN, and the Decoder. For each fault model, it shows the percentage of instructions that have different number of bits flipped at the output latch. The bits are binned on a log scale.

Table 3 shows that the corruption patterns of μarch-level faults for the ALU, AGEN, and Decoder are quite different from those of the gate-level faults. While μarch-level ALU and AGEN faults are injected in the output latches and corrupt at most one bit, the corresponding gate-level faults, though usually corrupt one bit, can result in multi-bit corruptions (between 9% and 25% across the ALU and the AGEN). However, for μarch-level Decoder faults, although faults are injected at the input latch, the resulting multi-bit corruptions turn out to be too aggressive (22% of corruptions for μarch-level faults are 8+ bits while the corresponding numbers for gate-level faults are less than 15%). This is because the output cone of the input (output) latch of the faulty unit is too large (small) when compared to that of a gate-level fault and leads to aggressive (conservative) bit corruptions at the output latch.

To better understand how the microarchitectural state gets corrupted by the injected faults, we collect the probability that bit i was flipped, given an instruction activates the underlying fault. Figures 5(a) and (b) show the distribution of the probabilities of a given bit in the output latch (numbered from bit 0 to bit 63) to be faulty under μarch-level stuck-at-0, gate-level stuck-at-0, and gate-level 1-cycle delay models for the ALU and the AGEN respectively. For brevity, we omit the μarch-level stuck-at-1, gate-level stuck-at-1, and 0.5-cycle delay models.

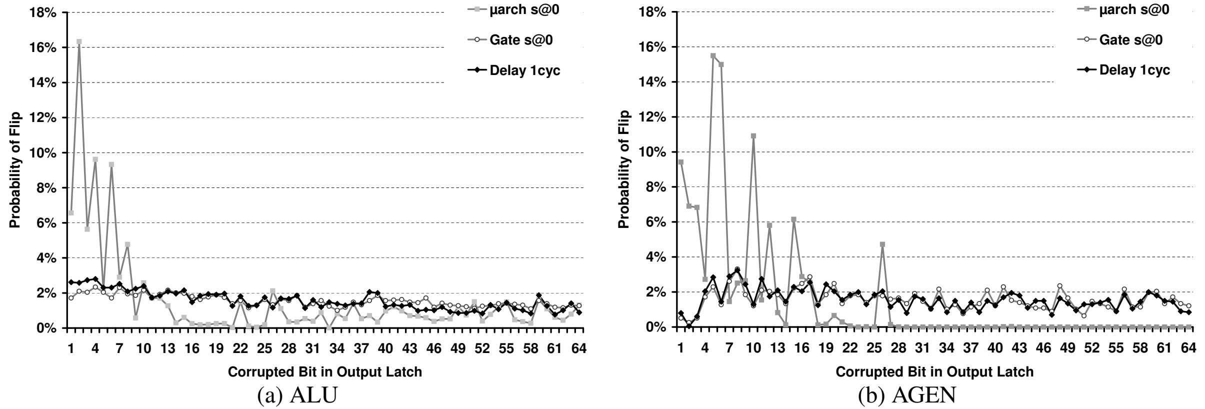
From the figures, we see that the probabilities of bit-flips of the μarch-level model are vastly different from the gate-level models. Further, the probability of flipping lower order bits is higher for μarch-level faults as the applications we use predominantly perform computations on the lower order 32-bits. The difference presented here is another source of discrepancy of the μarch-level model to represent gate-level faults.

When comparing the two gate-level fault models, interestingly, both have very similar corruption patterns even though they differ in terms of coverage, detection latency, activation rate, and number of bit-flips. To investigate this phenomenon, we studied the differences between corruption patterns of the gate-level stuck-at and delay fault injected at the same net and made the following observation: delay faults generally yield more corruption patterns than the stuck-at-0 faults because they can cause the same bit to be corrupted in both directions, instead of a single direction in stuck-at-0 faults. While this higher number of corruption patterns may cause delay faults easier to be detected, we note that the average activation rate of delay faults is also lower than that of stuck-at faults, as explained in Section 4.3.1, making them harder to detect and causing longer detection latencies.

Overall, our analysis shows that the different activation rates and bit corruption patterns paint a clearer picture in explaining the differences in the coverage (Figure 2) and the detection latencies (Figure 3) between μarch-level and gate-level faults. We found that higher activation rates of μarch-level stuck-at-1 faults typically cause higher coverage (and lower detection latencies) than gate-level faults, but it is not a perfect correlation. In some cases, despite significant differences in activation rates, the coverage of gate-level and

	ALU					AGEN					Decoder				
Bits	1	2	4	8	9+	1	2	4	8	9+	1	2	4	8	9+
μarch	100.0%	0.0%	0.0%	0.0%	0.0%	100.0%	0.0%	0.0%	0.0%	0.0%	72.5%	0.2%	4.8%	8.9%	13.4%
Gate s@1	91.1%	4.7%	1.2%	1.1%	1.9%	87.1%	6.8%	5.0%	1.0%	0.1%	66.1%	14.9%	10.5%	6.2%	2.3%
Gate s@0	84.4%	4.6%	2.8%	1.1%	7.1%	75.5%	8.4%	8.6%	7.4%	0.0%	60.8%	22.3%	12.2%	2.6%	2.2%
Delay 1cyc	90.4%	3.9%	1.4%	1.1%	3.2%	90.5%	4.1%	3.7%	1.5%	0.2%	71.7%	11.1%	12.5%	1.7%	2.9%
Delay 0.5cyc	75.0%	5.8%	2.2%	3.9%	13.1%	83.7%	7.9%	3.1%	2.4%	2.8%	68.2%	12.8%	4.3%	2.7%	12.0%

Table 3. Percentage of bits incorrect at the output latch.



(a) ALU (b) AGEN

Figure 5. Probability of corrupting each bit of the output latch, under μarch-level s@0, gate level s@0, and gate level delay models.

μarch-level faults is quite close. This is because once activated, gate-level faults cause different multi-bit corruption patterns. In some cases, these patterns are more intrusive than the μarch-level fault corruptions, boosting the coverage of the gate-level faults despite their lower rate of activation. In other cases, the higher intrusiveness of the multi-bit corruptions is not enough to compensate for the very low activation rates – this is specifically the case for gate-level delay faults which see the lowest coverage numbers.

We see that such complex interactions have a push-and-pull effect in determining the system-level outcome of faults and conclude that simple μarch-level stuck-at faults are inaccurate for modeling gate-level faults because they fail to (1) capture the system-level behavior, such as application-level masking, (2) induce different activation rates, and (3) accurately model μarch-level multiple bit corruption patterns. Therefore, any accurate μarch-level fault model for gate-level faults must account for all these factors to accurately capture their behavior.

4.4 Probabilistic μarch-Level Fault Models

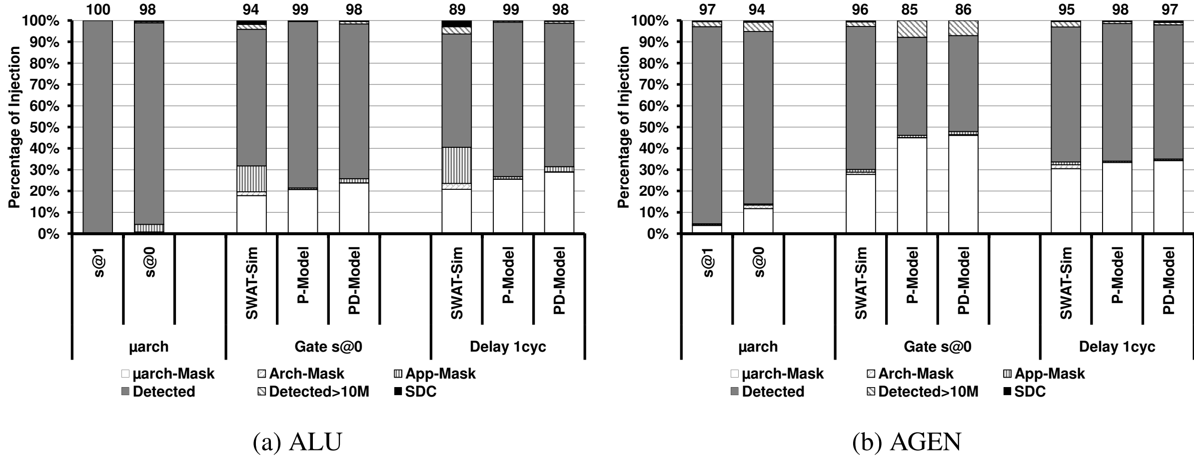
Given the inaccuracy of the μarch-level stuck-at fault model, we investigate whether we can derive alternate μarch-level fault models based on our analysis of the manifestation of the gate-level faults (both stuck-at and delay) at the microarchitecture. Such a model would be invaluable for accurately simulating the effect of the fault at the μarch-level, without invoking a gate-level simulator.

We investigated the behavior of the gate-level stuck-at and delay faults and found that each gate-level fault is activated differently and leads to different software-level outcomes. Hence, in our first-cut μarch-level fault model, we develop probabilistic models on a per-run basis, i.e., a different probabilistic model for each injected gate-level fault. In particular, we profile each SWAT-Sim run and collect the probabilities of the number of bits flipped at the output latch, the patterns of the flips, and the directions of the flips. Based on the collected information, we then derive two probabilistic μarch-level fault models, called the *P-model*, and the *PD-model*, respectively.

In the P-model, when an instruction uses the faulty unit, we decide on which bits to flip in the output latch based on the previously observed probabilities of the different number of bit-flips for this gate-level fault injection run (essentially using a table like Table 3, but built on a per-run basis). We then condition on this probability to decide on the pattern of the flip (similar to Figure 5 for different numbers of bit flips, but again on a per-run basis). All the bits indicated by this pattern are then flipped.

The PD-model refines the P-model by enforcing the direction of the bit-flips based on the profiling runs. That is, if the observed corruption pattern in the profiling run shows bit 3 of the output latch has a 1-to-0 (0-to-1) corruption, in the PD-model, this bit is corrupted only if it is a 1 (0).

We developed the P-model and the PD-model for both the gate-level stuck-at-0 and 1-cycle delay faults for the ALU and the AGEN. Figure 6 shows the ability of the P-model and the PD-model in mimicking the behavior of the corresponding gate-level fault models, evaluated using the coverage (similar to Figure 2). The number on top of each bar gives the coverage of the SWAT detectors for faults injected in that fault model. The results for gate-level stuck-at-1 and 0.5-cycle delay faults are not shown for the sake of clarity of the figures, and lead to similar conclusions as the other fault models.



(a) ALU (b) AGEN

Figure 6. The accuracy of the derived P- and PD-models for modeling gate level faults in (a) ALU and (b) AGEN, evaluated using the coverage of the SWAT detectors. The models closely mimic the masking outcomes (μarch-Mask+Arch-Mask) of the gate-level faults but do not, in general, accurately model their system level effects, resulting in differences in coverage.

From the figures, we see that both the P-model and the PD-model follow the μarch-level masking effects of the gate-level faults more closely than the μarch-level stuck-at faults. Nevertheless, the P- and PD-models for both gate-level stuck-at-0 and 1-cycle delay ALU faults are unable to capture the application-level masking effect while the two models for gate-level stuck-at-0 AGEN faults over-estimate the μarch-level masking effect.

In terms of coverage, the P- and PD-models do reasonably well for gate-level ALU stuck-at-0 fault and AGEN 1-cycle delay fault with differences less than 5%. However, for the other fault models, the P- and PD- models have 9+% differences in coverage.

In spite of extensive analysis and modeling, the probabilistic models do not accurately capture the μarch-level behavior of gate-level faults due to the following reasons.

- The models are oblivious to temporal variation in the corruption rates, i.e., both the models use the probabilities of injecting k-bit flips as an average rate across all instructions for injections on a given wire.
- The probabilities on which the models pick the number of bits to flip, the pattern of the bit-flips, and the direction of the bit flips are not conditioned on the fault-free value on which the patterns are applied. For example, although the pattern says that bit 1 should be flipped from a *1* to a *0*, if the original value of the bit is 0, no flips occur. Thus, there are fewer flips than what the model expects, which skews the probabilities.
- The profiling runs consider the output value but overlook the input value that activates the fault in the circuit and produces the corrupted output.

As previously discussed, we derive a different model for each faulty run in SWAT-Sim that simulates a different fault in the gate-level circuit. However, for an abstract evaluation and accurate prediction, a unifying model that generalizes the proposed per-run models must be built. Based on the stated limitations of the P- and PD- models, an accurate unified μarch-level model for the gate-level faults may be realizable. Nonetheless, until such a model is developed, SWAT-Sim remains an efficient platform for simulating and observing the system-level effects of gate-level faults.

5 Conclusions

With several μarch-level fault tolerance proposals emerging, models that accurately depict the μarch-level effect of gate-level faults become increasingly important.

This paper proposes SWAT-Sim, a hierarchical simulator that models only the faulty unit at gate-level accuracy with the rest of the system modeled at the μarch level. The fast and accurate nature of SWAT-Sim makes it possible to observe the system-level effects of the gate-level fault models. Using SWAT-Sim, we evaluate the differences between the manifestations of μarch-level and gate-level faults at the system level. We found the simple μarch-level stuck-at fault models to be, in general, inaccurate for capturing the system-level effects of gate-level faults. Based on an analysis of the causes for these differences, we derive two probabilistic μarch-level fault models for gate-level faults. However, the models fail to capture the complex manifestations of gate-level faults, resulting in inaccuracies.

The inaccuracies in the existing μarch-level stuck-at fault models and the absence of more accurate models motivate using simulators like SWAT-Sim to accurately model the μarch-level effect of gate-level permanent faults.

Acknowledgments

We would like to thank Pradip Bose from IBM and Subhasish Mitra from Stanford University for initial discussions on this work, Tong Qi for an initial version of the decoder module, and Ting Dong for help with statistical analysis.

References

[1] T. Austin et al. Opportunities and Challenges for Better than Worst-Case Design. In *ASP-DAC '05: Proceedings of the*

2005 conference on Asia South Pacific design automation, pages 2–7, New York, NY, USA, 2005. ACM.

[2] T. M. Austin. DIVA: A Reliable Substrate for Deep Submicron Microarchitecture Design. In *Proceedings of International Symposium on Microarchitecture*, 1998.

[3] J. Blome et al. Self-Calibrating Online Wearout Detection. In *Proceedings of International Symposium on Microarchitecture*, 2007.

[4] S. Borkar. Microarchitecture and Design Challenges for Gigascale Integration. In *Proceedings of International Symposium on Microarchitecture*, 2005. Keynote Address.

[5] F. A. Bower, D. Sorin, and S. Ozev. Online Diagnosis of Hard Faults in Microprocessors. *ACM Transactions on Architecture and Code Optimization*, 4(2), 2007.

[6] M. Bushnell and V. Agarwal. *Essentials of Electronic Testing for Digital, Memory, and Mixed-Signal VLSI Circuits*. Springer, 2000.

[7] H. Cha et al. A Gate-Level Simulation Environment for Alpha-Particle-Induced Transient Faults. *IEEE Transactions on Computers*, 45(11), 1996.

[8] K. Constantinides et al. Software-Based On-Line Detection of Hardware Defects: Mechanisms, Architectural Support, and Evaluation. In *Proceedings of International Symposium on Microarchitecture*, 2007.

[9] C. Dawson, S. Pattanam, and D. Roberts. The Verilog Procedural Interface for the Verilog Hardware Description Language. In *Verilog HDL Conference*, 1996.

[10] Z. Kalbarczyk et al. Hierarchical Simulation Approach to Accurate Fault Modeling for System Dependability Evaluation. *IEEE Transactions on Software Engineering*, 25(5), 1999.

[11] G. Kanawati et al. FERRARI: A Flexible Software-Based Fault and Error Injection System. *IEEE Computer*, 44(2), 1995.

[12] M. Li et al. Understanding the Propagation of Hard Errors to Software and Implications for Resilient Systems Design. In *Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems*, 2008.

[13] M. Martin et al. Multifacet's General Execution-Driven Multiprocessor Simulator (GEMS) Toolset. *Computer Architecture Newsletters*, 33(4), 2005.

[14] A. Meixner, M. Bauer, and D. Sorin. Argus: Low-Cost, Comprehensive Error Detection in Simple Cores. In *Proceedings of International Symposium on Microarchitecture*, 2007.

[15] S. Mirkhani, M. Lavasani, and Z. Navabi. Hierarchical Fault Simulation Using Behavioral and Gate Level Hardware Models. In *11th Asian Test Symposium*, 2002.

[16] B. C. Paul et al. Temporal Performance Degradation Under NBTI: Estimation and Design for Improved Reliability of Nanoscale Circuits. In *DATE*, 2006.

[17] A. Pellegrini et al. CrashTest: A Fast High-Fidelity FPGA-Based Resiliency Analysis Framework. In *International Conference on Computer Design*, 2008.

[18] M. Pirvu, L. Bhuyan, and R. Mahapatra. Hierarchical Simulation of a Multiprocessor Architecture. 2000.

[19] M. Prvulovic et al. ReVive: Cost-Effective Architecture Support for Rollback Recovery in Shared-Memory Multiprocessors. In *Proceedings of International Symposium on Computer Architecture*, 2002.

[20] R. Raghuraman. Simulation Requirements For Vectors in ATE Formats. In *Proceedings of International Test Conference*, 2004.

[21] P. Ramachandran et al. Statistical Fault Injection. In *Proceedings of International Conference on Dependable Systems and Networks*, 2008.

[22] S. Sahoo et al. Using Likely Program Invariants to Detect Hardware Errors. In *Proceedings of International Conference on Dependable Systems and Networks*, 2008.

[23] J. H. Saltzer et al. End-to-End Arguments in System Design. *ACM Trans. on Comp. Systems*, 2(4), 1984.

[24] S. Sarangi et al. A Model for Timing Errors in Processors with Parameter Variation. In *International Symposium on Quality Electronic Design*, 2007.

[25] S. Shyam et al. Ultra Low-Cost Defect Protection for Microprocessor Pipelines. In *Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems*, 2006.

[26] D. Sorin et al. SafetyNet: Improving the Availability of Shared Memory Multiprocessors with Global Checkpoint/Recovery. In *Proceedings of International Symposium on Computer Architecture*, 2002.

[27] J. Srinivasan et al. Exploiting Structural Duplication for Lifetime Reliability Enhancement. In *Proceedings of International Symposium on Computer Architecture*, 2005.

[28] Sun. OpenSPARC T1 Processor. Website, 2007. `http://www.opensparc.net/`.

[29] Virtutech. Simics Full System Simulator. Website, 2006. `http://www.simics.net`.

[30] N. Wang and S. Patel. ReStore: Symptom-Based Soft Error Detection in Microprocessors. *IEEE Transactions on Dependable and Secure Computing*, 3(3), July-Sept 2006.

[31] D. Yen. Chip Multithreading Processors Enable Reliable High Throughput Computing. In *Proceedings of International Reliability Physics Symposium*, 2005. Keynote Address.