CrashTest'ing SWAT: Accurate, Gate-Level Evaluation of Symptom-Based Resiliency Solutions

A. Pellegrini[1], R. Smolinski[2], L. Chen[2], X. Fu[2], S. K. S. Hari[2], J. Jiang[1], S. V. Adve[2], T. Austin[1], and V. Bertacco[1]

[1]Department of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor
[2]Department of Computer Science, University of Illinois at Urbana-Champaign

***Abstract*—Current technology scaling is leading to increasingly fragile components, making hardware reliability a primary design consideration. Recently researchers have proposed low-cost reliability solutions that detect hardware faults through software-level symptom monitoring. SWAT (SoftWare Anomaly Treatment), one such solution, demonstrated with microarchitecture-level simulations that symptom-based solutions can provide high fault coverage and a low Silent Data Corruption (SDC) rate. However, more accurate evaluations are needed to validate such solutions for hardware faults in real-world processor designs.**

In this paper, we evaluate SWAT's symptom-based detectors on gate-level faults using an FPGA-based, full-system prototype. With this platform, we performed a gate-level accurate fault injection campaign of 51,630 fault injections in the OpenSPARC T1 core logic across five SPECInt 2000 benchmarks. With an overall SDC rate of 0.79%, our results are comparable to previous microarchitecture-level evaluations of SWAT, demonstrating the effectiveness of symptom-based software detectors for permanent faults in real-world designs.

I. Introduction

Experts agree that future processor designs will suffer from increased rates of in-field hardware failures as a result of decreasing feature size [1]. Conventional solutions use heavyweight redundancy with high performance, area, and energy overheads, making them prohibitive for many processor designs. Recent work has explored lighter-weight solutions based on the insight that only faults that affect software behavior are problematic; faults that are masked at different system levels need not be detected [3, 5, 7, 10, 13]. These approaches therefore rely on monitors of anomalous software behaviors (*e.g.*, fatal traps and kernel panics) as symptoms of hardware faults. The SoftWare Anomaly Treatment (SWAT) project [5] represents the state-of-the-art in such systems. SWAT employs very low cost monitors for fault detection. In the infrequent case of a detection, SWAT triggers a more sophisticated fault diagnosis and checkpoint-based recovery procedure. While SWAT can detect most hardware faults, some faults may corrupt application outputs without triggering a SWAT detector. Such events, called silent data corruptions (SDCs), have been shown to occur for $< 1\%$ of hardware fault injections in microarchitectural simulations [5] making SWAT a promising reliability solution.

This work is supported in part by the Gigascale Systems Research Center (funded under FCRP, an SRC program), the National Science Foundation under Grant # CNS 0615005, CCF 0541383 and CCF 0811693 and Grant # 0937060 to the Computing Research Association for the CI Fellows project, and OpenSPARC Centers of Excellence supported by Sun Microsystems.

978-3-9810801-8-6/DATE12/©2012 EDAA

However, since microarchitectural simulators simplify many of the low-level design features of a processor, it is important to evaluate SWAT's detectors on a realistic system to validate the results obtained through less detailed models. An accurate evaluation of SWAT needs to meet three requirements: **(1)** gate-level hardware modeling, since real faults manifest at the gate level and can impact software behavior differently from microarchitecture-level fault models, **(2)** the ability to inject hardware faults across the entire processor design to account for differences in the behavior of hardware components (*e.g.*, microarchitecture-level simulations typically do not model control paths), and **(3)** the ability to quickly run millions of instructions and a full-system software stack (e.g., OS and application) because these layers can influence fault propagation and thus software behavior.

Meeting these criteria with software simulators is impractical due to the amount of computation required to simulate a full hardware design for millions of cycles. Gate-level software simulations run at tens of cycles per second [11]. To improve simulation speed, SWATSim [6], a hybrid microarchitecture/gate-level simulator, was developed to allow for gate-level fault injection experiments at the speed of microarchitecture-level simulations by modeling just the faulty component with gate-level accuracy. However, this simulator could lose evaluation accuracy at the interfaces between microarchitecture-level and gate-level components. Additionally, the evaluation in SWATSim was limited to three hardware components (address generator, arithmetic logic unit, and decoder); extending the framework to a full processor is difficult [6].

An approach to achieve higher accuracy, speed, and processor coverage is to use reconfigurable hardware, such as Field-Programmable Gate Arrays (FPGAs), to emulate faults in digital designs. Previous work [2, 9, 11] performed gate-level fault injections on a processor design mapped on an FPGA, but their evaluations did not study the effects of hardware faults on software with a full OS and application running. This work develops a comprehensive and high performance FPGA framework to study gate-level hardware faults on a real-world system running a complete full-system software stack. We built our platform on the OpenSPARC project [12], augmented with SWAT detectors and the CrashTest infrastructure [11] to allow for fault injection experiments with detailed fault models without hindering performance or accuracy.

CrashTest [11] automatically instruments a digital design with the logic necessary to model hardware faults at the gate level. It takes as input the design's RTL and the number of

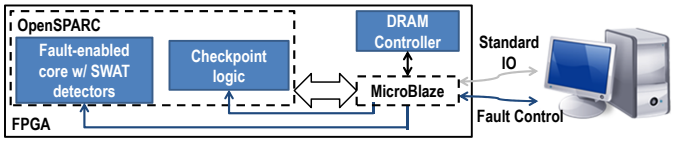

Fig. 1: Experimental setup and design modifications.

faults to inject, and it automatically injects hardware faults leveraging accurate fault models. It then maps the fault-enabled design to a hardware emulation platform. It injects multiple fault sites for each synthesized netlist, as the last step for FPGA mapping typically requires a considerable amount of time; however, each fault site can be individually enabled. CrashTest can accelerate the resiliency analysis of industrial-size designs by up to six orders of magnitude compared to equivalent software simulations [11].

Our final platform (OpenSPARC + SWAT detectors + CrashTest on an FPGA) meets all three of the above-mentioned evaluation requirements for SWAT by allowing us to perform accurate gate-level evaluation of hardware faults across the entire processor design and run complete applications on top of a full OS and hypervisor.

We used this platform to evaluate the effectiveness of SWAT's detectors with a total of 30,800 stuck-at and 20,830 path-delay fault injection experiments across five SPECInt 2000 benchmarks. Overall, these experiments validate the results previously reported by software-based simulations of SWAT, but also reveal some interesting differences: **(1)** a high fault masking rate (62.56% of experiments); **(2)** silent data corruptions (0.79% of experiments) are concentrated within a handful of functional units in the processor data path; **(3)** the range of software symptoms detected is much wider than recognized in previous evaluations of SWAT.

II. Evaluation Platform

Our FPGA platform builds on the OpenSPARC T1 project [12] from Sun Microsystems and is mapped onto a Xilinx Virtex-5 FPGA. The FPGA design includes a processor core based on the UltraSPARC T1 design and a Xilinx MicroBlaze core that handles processor features not implemented in the OpenSPARC core. The platform runs the OpenSolaris operating system, the hypervisor, and SPARC-V9 applications. Our platform was modified to include the following features for fault injection experiments on the system. The core was instrumented with CrashTest to emulate faulty gates. A checkpoint and restore system was added to reduce setup time (Section II-A). Several SWAT detectors were implemented in hardware (Section II-B). We added logic for a second serial port and a control channel between the MicroBlaze and OpenSPARC core to enable CrashTest fault logic. The DRAM controller was enhanced to support higher memory capacities.

Figure 1 shows a high-level representation of our experimental setup, with the major design modifications highlighted by a darker tone. Our fault-enabled OpenSPARC core on the FPGA has a clock rate of 10MHz, a six orders of magnitude speedup compared to software simulations with equivalent fault accuracy. Our software modifications to the OpenSPARC distribution were confined to the MicroBlaze firmware. The firmware was modified to allow for fault injection control and to perform the checkpoint and restore operations.

A. Checkpoint and Restore System

In the original platform, OS boot time is a major bottleneck in performing the large set of experiments required for a statistically accurate evaluation. In order to avoid this large setup time, we implemented a checkpoint and restore mechanism for the OpenSPARC T1. With this system, the time required for a single board setup is shared among several fault injections, significantly reducing total experiment runtime.

The checkpoint operation copies the processor architectural state to shadow registers added to the OpenSPARC processor design, and the rest of the system state is saved into a reserved section of DRAM. The MicroBlaze firmware ensures that these parts form a consistent checkpoint. A restore operation rolls back to the checkpointed state.

B. Fault Detectors

We evaluated a variety of fault detectors inspired by the SWAT system, customizing them to fit the OpenSPARC design. In a full SWAT implementation, most of these detectors would be implemented through software or firmware traps (for this work, there was insufficient support to recompile the OS and hypervisor with our modifications).

Fatal Traps and Kernel Panics: Previous SWAT work [5] reports that these detectors are commonly invoked in the presence of faults. Fatal traps include traps due to events such as divide-by-zero, misaligned accesses, maximum trap level reached and kernel panics. We monitor for these two latter events in hardware.

Hypervisor Crashes: Error outputs to the console from the hypervisor are considered as successful fault detections. An example is a TLB miss at the hypervisor privilege level. In a real system, these cases would trap to the firmware diagnosis/recovery at the point of failure detection and before sending the error message output.

Firmware Checks: These detectors are triggered by failing checks in the firmware running on the MicroBlaze. We report failed checks as fault detections since these events (e.g. out-of-bounds addresses for memory operations, invalid I/O requests) cause the firmware to abort execution.

Hardware Stalls: We detect a fault if a hardware thread has not issued instructions for a period longer than a predefined threshold of 300 million cycles. This high time limit was selected to eliminate any false positives.

Application Abnormal Exits: These symptoms are detected by monitoring the standard output of the benchmark and they include the following application outcomes: segmentation fault, core dump, dynamic linker errors, error messages from the operating system, abnormal program termination and program assertion failures. Again, in a real system, these symptoms would invoke diagnosis/recovery, and not result in an erroneous standard output message.

SWAT detectors not included: Others SWAT detectors not included here are a hang detector and a high OS detector [5].

III. Methodology

A. Fault Locations

We injected stuck-at and path-delay faults in random nets in the design. To meet timing constraints, the core was partitioned into multiple modules with each module having a number of fault locations proportional to its area. The area

TABLE I: OpenSPARC modules injected with faults.

OpenSPARC T1 module	Gate count	FF count	Stuck-at Faults	Path-delay Faults
Arithmetic Logic - ALU	1,968	65	19	9
Divide - DIV	3,277	486	31	65
Error Corr. and Ctl. - ECC	998	237	10	32
Execution Control - ECL	1,727	335	17	45
Float. Point FE - FFU	5,776	836	55	112
Instruction Fetch - IFU	13,980	3,775	225	511
Load Store - LSU	24,127	4,397	635	594
Multiplier - MUL	14,665	647	138	87
Reg. Management - RML	1,206	231	11	31
Reg. Bypass Logic - BYP	5,938	708	56	95
Shift - SHFT	1,767	0	9	0
Trap Logic - TLU	18,693	3,737	334	502

was approximated by the number of gates in the module's gate-level netlist, and the total number of fault locations was computed for a confidence level of 95% and a confidence interval of 4%. Faults were injected in the control logic of memory arrays but not in their storage elements, since those can be protected by error correcting codes. Table I lists the fault enabled modules, the total number of gates in each module, and the number of different fault locations within each module.

B. Benchmarks

We evaluated the effects of stuck-at and path-delay fault models on five applications (mcf, vpr place, parser, vortex, and twolf) from the SPECInt 2000 benchmark suite with the test or reduced input sets [4]. Using smaller input sizes was a necessity since every fault that is not detected requires that the application runs to completion to determine if the fault has silently corrupted the application. To run the 50,000 faults with the reference inputs would require an expected FPGA run-time of 40 years. All benchmarks were compiled for the SPARC-V9 architecture with $-O3$ optimization.

C. Fault Injection Experiments

For our experiments, we performed fault injections in two application execution points: the first one is immediately after benchmark initialization, the second is approximately halfway through its execution. A restore operation is performed after each fault injection. After each restore, we allow 5 seconds (50 million cycles) for the system to rewarm the caches and populate its TLB. Experiments that do not trigger any fault detector require executing the benchmark until completion to determine if the fault corrupted the application outputs. For this work, we activated a single fault per experiment.

IV. Results

For our evaluation, we performed 30,800 stuck-at and 20,830 path-delay fault injection experiments across all modules of the OpenSPARC T1 core. For each fault experiment, we monitored the behavior of the system and categorized the possible outcomes into five mutually exclusive categories that we discuss in detail below. Figure 2 shows the outcome of the stuck-at-0, stuck-at-1, and path-delay experiments.

A. Masked

This category is for experiments that completed the execution with correct output. We observed a high masking rate of 60.7% for stuck-at and 65.3% for path-delay fault. These results are higher than previously observed in microarchitectural-level permanent fault injections [5] (16%) and the hybrid mircoarchitectural/gate-level simulation results for three modules simulated with SWATSim (30% to 40%) [6]. Following are several factors that we believe contributed to this difference:

1) The OpenSPARC core was originally designed to support four hardware thread contexts. However, limited FPGA resources constrained us to using the one threaded version of the OpenSPARC T1 core, which has some design details leftover from the original multi-threaded processor design. These unused components are in the synthesized design and are candidates for fault injection sites, thus potentially increasing the overall masking rate.
2) A few modules, such as the multiplier, the floating point frontend, and the trap logic unit, contain circuitry that is not exercised frequently, if at all, by our applications.
3) Our infrastructure models design aspects not previously simulated (such as gate-level characteristics of the design), thus adding extra layers of masking that could prevent injected faults from affecting application outputs.
4) Previous fault injection campaigns performed through SWATSim only analyzed a part of the design, not testing three large modules – instruction fetch, load-store unit, and trap handling logic– which all have a masking rate above 55% for both stuck-at and path-delay experiments.

The ALU is a particularly interesting case, since our experiments produced a much lower masking rate than the one observed for stuck-at and path-delay faults with SWATSim (up to 40%). We explain this difference by the fact that OpenSPARC uses the ALU for both address generation and normal arithmetic integer operations, whereas SWATSim modeled a core with separate address generation and arithmetic/logic units.

B. Detect

We declare a fault to be detected when one of our detectors (Section II-B) is triggered during application execution. The overall detection rate is 30.4% and 29.4% for stuck-at and path-delay fault models, respectively. Table II shows the percentage of detections that each detector is responsible for. We found that a large portion of hardware stalls, roughly 54%, were due to hardware faults injected in control logic in the load-store unit. Also, 38.9% of detections are kernel panics due to faults in the datapath submodules of the load-store unit. Overall, the types of detections occurring in the OpenSPARC platform compose a larger variety than the ones reported in previous SWAT evaluations due to unexpected application or OpenSolaris services failure.

C. Timeout

About 6.4% of our fault injections experiments reached our simulation time limit of twice the expected fault-free run-time for an application. 7.8% and 4.2% of the experiments for stuck-at and path-delay timed out, respectively. We believe that most of these cases are erroneous infinite loops at some level of the software stack, but we were unable to clearly identify them due to the lack of an on-line hang detector in

TABLE II: SWAT fault detector breakdown.

Fault Type	Kernel Panics	Fatal Traps	Firmware Checks	Hypervisor Crashes	Abnormal Exits	Core Stalls
Stuck-at	31.7%	25.0%	11.0%	9.6%	6.8%	15.9%
Path-delay	44.7%	20.4%	4.7%	3.6%	9.0%	17.6%

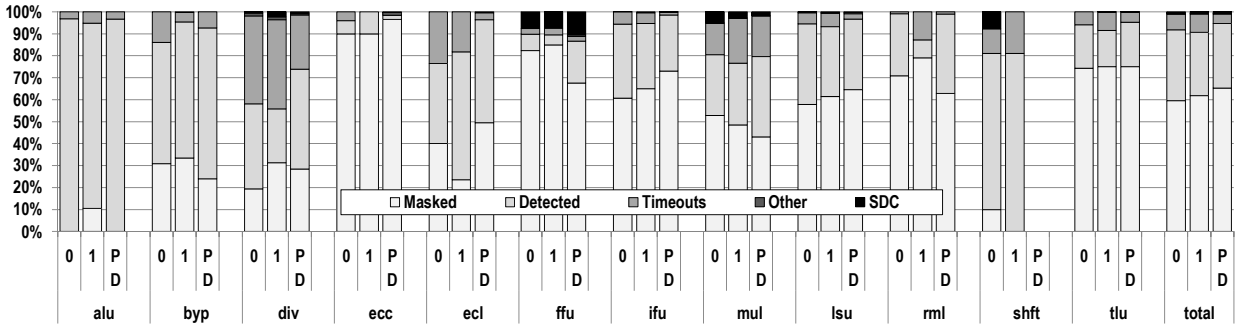

Fig. 2: Breakdown of experiments for stuck-at-0 (0), stuck-at-1 (1), and path delay (PD) faults. The x-axis indicates the OpenSPARC module studied, while the y-axis reports the percentage of fault injections resulting in a given outcome.

our platform. Future work will look at implementing on-line hang detectors similar to those presented in [8].

As a first step to understanding the timeout cases, we developed a feature to allow us to trace the currently retiring program counters (PCs). We performed this logging only for path-delay experiments due to time limitations. Our logging feature allows us to record roughly 1K PCs in a minute. In order to detect infinite loops, we relied on sampling batches of 1K PCs across six different time slots in a $\sim$30 million instruction window. With these records, we could determine if the PCs logged in a sample period are a subset or a superset of all other sample periods. For simplicity, we removed the PCs corresponding to hypervisor interrupts. With this analysis, we found that 274 out of 867 experiments that timed-out are likely to be hangs. Classifying the other cases was infeasible due to our sampling technique and the interference of OS and hypervisor services running in the background.

D. Other Anomalous Outcomes

About 0.27% of our experiments were placed into the "Other" category (0.2% of stuck-at experiments and 0.4% of path-delay experiments). A fraction of these cases were due to erroneous software behavior that caused the file system to become unusable or full. In a deployed machine, the file system free space will typically be much larger than on our experimental platform, so it is unclear how these failures would affect a real system. Additionally, there are a number of cases where we could not interact with the standard input and output interface of the machine for a few hundred seconds.

E. Silent Data Corruptions

The last outcome is for experiments where the application finishes and the application output differs from the expected application output; *i.e.*, a silent data corruption (SDC). Our overall SDC rate is 0.82% for stuck-at faults and a 0.75% for path-delay faults.

Interestingly, we found that all but four units produced none to very few SDCs (0 SDCs for ALU, ECL, and RML; under 0.4% SDC rate for BYP, ECC, IFU, LSU, and TLU). SHFT, DIV, and MUL had higher SDC rates, but under 5%. The FFU had the highest SDC rate at 7.55% for stuck-at faults and 10.47% for path-delay faults. Thus, the vast majority of the SDCs are concentrated in units that compute data values for the program (as opposed to addresses or control related operations) – these should be the focus of further investigations to improve reliability.

V. Conclusions and Future Work

This paper tested the effectiveness of low-cost fault detectors proposed by SWAT on an industrial-strength microprocessor core with an extensive number of gate-level permanent fault injections. We injected a total of 30,620 stuck-at faults and 20,830 path-delay faults throughout the logic of the OpenSPARC core. The current set of SWAT detectors were able to detect 80.2% of the unmasked faults, and many of the remaining undetected cases may be application or OS hangs. Overall, a very small fraction (0.79%) of the experiments led to silent data corruptions. These results are significant since they constitute a crucial step towards validating SWAT and other symptom-based detection solutions in general, for use as a reliability solution in industrial hardware.

In the future, we would like to extend this work by studying more fault models and more benchmarks.

References

[1] S. Borkar. Designing Reliable Systems from Unreliable Components: The Challenges of Transistor Variability and Degradation. *IEEE Micro*, 25(6), 2005.

[2] P. Civera et al. FPGA-Based Fault Injection Techniques for Fast Evaluation of Fault Tolerance in VLSI Circuits. *Lecture Notes in Computer Science*, 2147, 2001.

[3] M. Dimitrov and H. Zhou. Unified Architectural Support for Soft-Error Protection or Software Bug Detection. In *Proc. PACT*, 2007.

[4] A. J. KleinOsowski and D. J. Lilja. MinneSPEC: A New SPEC Benchmark Workload for Simulation-Based Computer Architecture Research. *IEEE Computer Architecture Letters*, 1, 2002.

[5] M. Li et al. Understanding the Propagation of Hard Errors to Software and Implications for Resilient Systems Design. In *Proc. ASPLOS*, 2008.

[6] M. Li et al. Accurate Microarchitecture-Level Fault Modeling for Studying Hardware Faults. In *Proc. HPCA*, 2009.

[7] G. Lyle et al. An End-to-end Approach for the Automatic Derivation of Application-Aware Error Detectors. In *Proc. DSN*, 2009.

[8] N. Nakka et al. An Architectural Framework for Detecting Process Hangs/Crashes. In *Proc. EDCC*, 2005.

[9] S. Nomura et al. Sampling + DMR: Practical and Low-overhead Permanent Fault Detection. In *Proc. ISCA*, 2011.

[10] K. Pattabiraman et al. Dynamic Derivation of Application-Specific Error Detectors and their Implementation in Hardware. In *Proc. European Dependable Computing Conference*, 2006.

[11] A. Pellegrini et al. CrashTest: A Fast High-Fidelity FPGA-based Resiliency Analysis Framework. In *Proc. ICCD*, 2008.

[12] Sun Microsystems Inc. OpenSPARC T1. http://opensparc-t1.sunsource.net/, 2005.

[13] N. Wang and S. Patel. ReStore: Symptom-Based Soft Error Detection in Microprocessors. *IEEE Trans. on Dependable and Secure Comp.*, 3(3), 2006.