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, swat@illinois.edu

***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 monitoring software-level symptoms. SWAT (SoftWare Anomaly Treatment), one such solution, demonstrated through microarchitecture level simulations that it can provide high fault coverage and a Silent Data Corruption (SDC) rate of under 0.5% for both permanent and transient hardware faults for all but one hardware component studied. More accurate evaluations of SWAT require tests on industry strength processor, a commercial operating system, unmodified applications, and accurate low-level fault models.**

In this paper, we propose a FPGA based evaluation platform that provides the software, hardware, and fault model accuracy to verify symptom-based fault detection schemes. Our platform targets a OpenSPARC T1 processor design running a commercial operating system, OpenSolaris, and leverages CrashTest, an accurate gate-level fault analysis framework, to model gate-level permanent faults. Furthermore, we modified the OpenSPARC core to support hardware checkpoint and restore to make a large volume of experiments feasible.

With this platform we provide results for 30,620 fault injection experiments across the major components of the OpenSPARC T1 design and running five SPECInt 2000 benchmarks. With a conservative, overall SDC rate of 0.94%, the results are similar to previous microarchitectural level evaluations of SWAT and are encouraging for the effectiveness of symptom-based software detectors.

I. Introduction

Shrinking feature sizes threaten higher runtime failure rates in future commodity systems, motivating low-cost resiliency solutions [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 not all hardware faults are problematic [2, 3, 4, 5, 6, 7, 8]. These approaches are based on the observation that, for most applications, only the faults that visibly affect software behavior should be treated. Faults that are masked at the circuit, microarchitectural, architectural, operating system, or application levels do not require any corrective action, and therefore, there is no advantage in detecting them. This approach detects hardware faults by monitoring for anomalous software behavior using very low cost monitors. In the infrequent case of a fault detection, the software symptom detection triggers a more sophisticated diagnosis and checkpoint-based recovery. The SWAT (SoftWare Anomaly Treatment) [3, 9, 10] system represents the state-of-the-art in such an approach. Evaluations of SWAT through software simulation at the microarchitectural level demonstrate the effectiveness of this approach. Such results - collected through randomized injection of permanent and transient faults in a core running various workloads - revealed a silent data corruption (SDC) rate lower than 0.5

Unfortunately, acceptably accurate evaluation of resiliency solutions that rely on hardware and software mechanisms (such as SWAT) remains challenging. Such evaluations require executing the complete software stack, consisting of long executions of applications on top of an operating system. Fault manifestations at the software level may span millions of cycles and assessing software masking or data corruptions may require running the application until completion. Furthermore, accurate modeling of hardware errors requires very detailed fault models and low-level knowledge of the design under evaluation. Unfortunately, software solutions capable of simulating complete computer systems in such detail are extremely slow (up to tens of cycles per second) [11] and thus are impractical to study fault effects on execution windows of millions or even billions of cycles. Most previous evaluations therefore adopted microarchitecture-level software simulations [8, 3]. Such simulations achieve viable performance for fault injections by heavily simplifying the hardware fault model. Modeling hardware faults at such high level causes reliability analysis to neglect important system characteristics such as control signals and circuit-level masking effects. The SWATSim approach [3] proposed a compromise between speed and accuracy by using a mixed gate-level/microarchitectural software simulation. Unfortunately, its use is constrained by the requirement to simulate and interface both microarchitectural and gate-level models of each component. (The SWATSim work interfaced only three components in this way.)

An alternative to software-based fault simulations is to employ reconfigurable hardware such as Field-Programmable Gate Arrays (FPGA) to accelerate fault injections. Previous works restricted their fault models to transient faults or were applicable only to very simple circuits [12, 13, 14]. In contrast, CrashTest is a resiliency analysis framework that addresses both fault model accuracy and fault simulation performance by providing an automated way to inject a variety of fault models, including permanent and transient faults, on complex

systems [11]. Through the use of FPGAs, CrashTest has been successfully used to evaluate the reliability of industrial-size designs without compromising the accuracy of the fault models.

In this paper, we use CrashTest to evaluate the effectiveness of SWAT to detect the permanent faults inserted in an industrial-strength processor core, the OpenSPARC T1 [15]. We modified the processor core and system's firmware to support a subset of SWAT detectors, and implemented a full-system checkpoint mechanism. We then injected 30,620 stuck-at faults across all hardware structures of the processor core. Each fault was activated at runtime, while the design was executing applications selected from the SPECInt 2000 benchmark suite within the environment provided by OpenSolaris, a commercial operating system. For each fault injection, we determine if its effects on the software benchmark were masked, caused a silent output corruptions (SDC), or whether the SWAT detectors were able to detect the hardware failure.

Overall, our FPGA-based experiments validate the results previously reported by software-based simulations of SWAT, but also reveal some interesting differences. First, the masking rate we report in this work is higher than the one reported in previous evaluations of SWAT performed on less accurate hardware and fault models. Second, the experiments that resulted in silent data corruptions were concentrated within a handful of hardware components – mostly complex functional units such as the floating point unit, the multiplier, and the divider. Finally, the range of software anomalies detected is much wider than previously recognized.

To the best of our knowledge, this work is the first to evaluate and validate the effectiveness of lightweight fault detection techniques for permanent faults on a commercial processor executing real applications on a commercial operating system through gate-level fault injections in all components of a processor core.

II. Experimental Methodology

A. FPGA Platform

Our FPGA framework is inspired by the OpenSPARC Project at Sun Microsystems [15], where a OpenSPARC T1 processor was mapped on to a Xilinx Virtex-5 FPGA. In this setup, a single processor core from the OpenSPARC T1 design and its L1 caches are implemented on the FPGA, while the L2 cache, memory controller and other basic peripherals are emulated through the support of an ancillary microprocessor mapped on the FPGA device (a Xilinx MicroBlaze).

The OpenSPARC core was instrumented with logic to emulate fault models in various locations through CrashTest (Section II-B). The MicroBlaze processor already present in the design was adopted to activate each fault location at runtime. Support for the SWAT detectors was also added in hardware(Section II-C). Significant modifications to the original design were also needed to modify the memory controller to increase the size of the off-chip DRAM. The extra memory space was used to store the benchmarks used in our experiments and the checkpointed memory. Finally, an additional communication channel between the host and the on-board MicroBlaze was established to allow runtime enabling and disabling of the fault locations. Figure 1 shows a high-level representation of our experimental setup. In the figure, the major design modifications are highlighted with a darker tone.

Transferring all the necessary data (FPGA configuration and memory images) from a host to the target FPGA board used in our experiments takes approximately 20 minutes. This design can successfully execute an unmodified version of the Sun OpenSolaris operating system, but an additional 50 minutes are necessary for the emulated machine to boot to an interactive console. To avoid spending such a large amount of time to setup the system for a single fault injection and to enable a large volume of experiments, we implemented a checkpoint and restore mechanism. With the checkpoint system, the time required for a single board setup can be shared among several fault injections (Section II-E). The checkpoint operation copies the processor architectural state (e.g. register files, PCs, trap stack) to shadow registers within the OpenSPARC processor design and also conservatively copies the entire processor memory space and file system into a shadow memory area. Processor checkpoint data is stored within the OpenSPARC design, and the memory checkpoint is handled by the MicroBlaze firmware. Thus, a sophisticated synchronization mechanism is necessary to coordinate these two checkpoints to take a coherent snapshot of system's status. A restore mechanism, which roll backs the state of the system to a certain checkpoint state stored in the system, has also been developed. In order to avoid issues with in-flight memory requests, we delay checkpoint operations until all load/store queues have been emptied and no active instructions are in the pipeline. Additionally, we invalidate the L1 caches and the TLBs during both checkpoint and restore, and the entire processor is reset before performing a restore to clear any leftover non-architectural state from the previous fault injection experiment.

B. CrashTest

CrashTest is able to automatically instrument a digital design with logic that mimics hardware faults at the gate level [11]. It takes as input the RTL of the design under evaluation and automatically injects faults leveraging accurate fault models. To maintain high evaluation speed without compromising fault accuracy, the fault-enabled design is mapped to a hardware emulation platform (FPGA). In generating a FPGA-ready fault-enabled system, CrashTest performs four transformations: 1) the original design is synthesized through Synopsys Design Compiler to produce a gate-level netlist; 2) the produced netlist, comprising only basic gates, is analyzed by CrashTest to identify possible fault locations; 3) logic that mimics fault behavior at the gate-level is inserted in selected locations of the netlist; 4) the complete system is finally mapped to the targeted FPGA device.

Multiple faults are injected into each synthesized design as this last step typically requires a considerable amount of

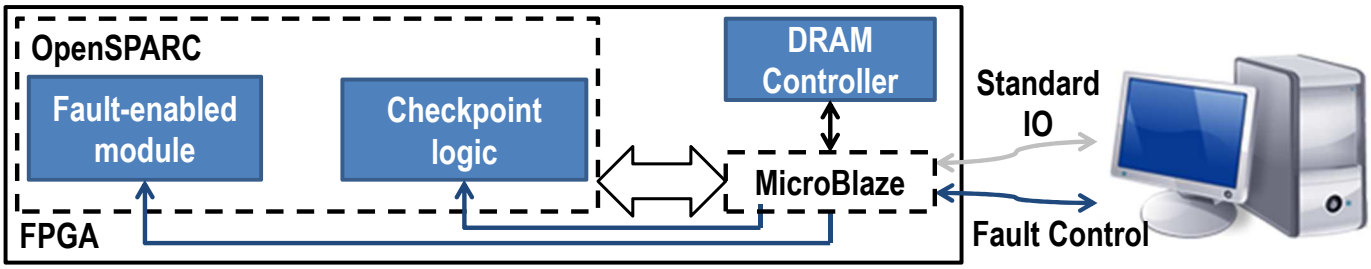

Fig. 1. Experimental setup and design modifications

time particularly due to the synthesis and place-and-route algorithms. Fault locations, when deactivated, do not alter the behavior of the design. Each fault location can also be individually activated to produce the effects of the fault. This approach allows CrashTest to amortize design setup time among several experiments.

CrashTest can accelerate resiliency analysis of industrial-size designs by up to six orders of magnitude compared to equivalent software-based fault injections [11]. Moreover, CrashTest does not alter the original design functionality, allowing it to execute a complete software stack, including the operating system and user applications. This characteristic is fundamental in testing the effectiveness of hybrid hardware/software fault tolerant solutions like SWAT.

For this work, due to area limitations in the FPGA device in our experimental setup, we could not inject faults throughout the entire OpenSPARC core at once. We therefore partitioned the design into multiple microarchitectural modules and injected faults in each of them (Section II-E). The timing achievable for the fault-enabled OpenSPARC core on our FPGA device is 100ns, enough to run the design at a frequency of 10MHz. Even though this frequency is four times slower than the one reached by the original design, it still yields a speed-up of six orders of magnitude compared to a software-based fault simulation with equivalent accuracy.

C. SWAT Detectors

We considered the following detectors based on the SWAT philosophy, customized for the OpenSPARC platform. In a real system, the detected symptoms would trigger a trap to the firmware which would invoke the diagnosis [10] and recovery mechanisms [16, 17].

Fatal Traps and Kernel Panics: Previous SWAT work 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, and maximum trap level [3]. The kernel panic detector is triggered when the OS enters its panic routine. We could not recompile the version of the OpenSolaris kernel provided with the FPGA platform to implement these detectors in software. Instead, we monitored the traps and kernel panic program counters in the hardware to trigger these detections.

Hypervisor Crashes: We use error messages printed by the hypervisor as a detection (again, we could not modify the source to catch the symptom before the message is printed). In our experiments, an example failure is a TLB miss exception that occurs at an invalid trap level.

Firmware Checks: The OpenSPARC firmware runs on the MicroBlaze to emulate the L2, packet receiving logic, and the memory controller. It performs a variety of consistency checks as part of its communication with the OpenSPARC core (many of them would be performed in hardware in a real machine). We report failed checks as fault detections since they would originally cause the firmware to abort execution. Examples include out-of-bounds addresses for loads and stores, and invalid request types from the core.

Hardware Stalls: We detect a fault if a hardware thread has not issued instructions for a period longer than a predefined threshold (set to 300 million cycles, or about 30sec, to avoid false positives).

Abnormal Exits: These symptoms were detected via console output monitoring and include the following: segmentation fault, core dump, dynamic linker errors, errors from OpenSolaris services, abnormal program termination, and program assertion failures. In all of these cases, a real system would trap to the firmware diagnosis/recovery at the point of failure detection and before sending the error message output.

SWAT detectors not included: The main SWAT detectors not included here are a hang detector and a high OS detector [3] and will be the focus of future work.

D. Workloads

We evaluated the effects of stuck-at faults in five applications extracted from the SPECInt 2000 benchmark suite with a combination of the test and reduced input sets [18] (Table I). We selected smaller input sets due to the large runtime ($>$ 1 hour) on the FPGA platform of the reference input sets. Since our experiments consisted in testing the effects of more that 30,000 faults, running the reference input set for such benchmarks was not a practical option. All benchmarks were compiled for the SPARC-V9 architecture with default ($-O3$) optimizations.

E. Fault Injections and Outcomes

For this paper, we only focused on injecting stuck-at faults in various nets in the design (we studied both stuck-at-1 or stuck-at-0 faults). As previously explained, we partitioned the core into multiple modules and injected faults in random locations in these modules. Table II lists the units into which

TABLE I
WORKLOADS

Benchmarks	Input Set	Number of Instructions	FPGA Time
175.vpr (place)	medium reduced	458M	9m 9s
181.mcf	test	419M	5m 27s
197.parser	medium reduced	913M	6m 16s
255.vortex	medium reduced	547M	11m 55s
300.twolf	test	415M	5m 15s

TABLE II
MODULES OF THE OPENSPARC INJECTED WITH FAULTS

OpenSPARC T1 unit	Gate count	Fault locations
Arithmetic Logic Unit (ALU)	1,968	19
Divide (DIV)	3,277	31
Error Correction and Control (ECC)	998	10
Execution Control Logic (ECL)	1,727	17
Multiplier (MUL)	14,665	138
Register Management Logic (RML)	1,206	11
Register Bypass Logic (BYP)	5,938	56
Floating Point Frontend Unit (FFU)	5,776	55
Instruction Fetch Unit (IFU)	13,980	225
Load Store Unit (LSU)	24,127	635
Trap Logic Unit (TLU)	18,693	334

faults were injected, the total number of gates in each unit, and the number of different fault locations that were used within each unit. The targeted number of faults injected in each module is a function of its area (approximated by the number of gates in the module's gate-level netlist) and was computed for a confidence level of 95% and a confidence interval of 4% (Table II). For three hardware units, the IFU, LSU and TLU, the ratio between the number of faults and number of gates is higher than for the other modules. This is due to the fact that these threes units, once instrumented with the checkpoint mechanism and faults, could not meet timing requirements on the FPGA hardware. Thus, we had to partition these three units into smaller sub-modules that were instrumented separately. For each of these three modules, the total gate count for the submodules is higher than for the original unit since the synthesizer has a narrower optimization scope. Note, however, that the increased ratio only increases the confidence of our results for the experiments performed on the IFU, LSU and TLU. No faults were injected in the memory array structures of the design (such as register file, caches, and TLBs) since these structures are protected with ECC or parity.

Experiments were run for each fault location in two different phases of the five selected benchmarks. The first fault injection point is roughly after the initialization portion of the benchmark, and the second point is roughly halfway between the first injection point and benchmark completion. For each of the two fault activation points, a checkpoint is taken before starting a fault injection campaign. We then wait for approximately 50 million cycles (5 seconds) to allow caches and TLBs to warm up, and then activate a single fault location. We then monitor the system to determine if any of the following termination conditions are met:

Detection: The fault is detected with one of the SWAT detectors previously described.

Masked: The application finishes without a detection and its output matches the golden application output.

Silent Data Corruption (SDC): The application finishes without a detection, but its output files differ from the golden output. Our definition of SDC is conservative – many of the outcomes differ in ways that are not important to the user (i.e., the fault is really masked) while others clearly show erroneous behavior (i.e., the fault is detectable by the user but may not be recoverable). Further analysis is needed to separate such instances and will be the focus of future work.

Timeout: To limit the experiment time, we declare a timeout if an injection experiment takes more than 150% of the time taken to run on the FPGA without faults injected. We expect many of these cases to be detected by a hang detector.

Other: This category includes cases where the application did not terminate in a normal way due to some idiosyncrasies of our current system (e.g., the file system was too small and filled up, a detector was triggered after the fault was deactivated, the application output printing hung in a way that was hard to explain). Further study is necessary to further understand such erroneous behaviors.

After the outcome of the experiment is known, the processor and memory state are restored to our checkpointed state and we continue with the next fault experiment.

III. RESULTS

Our experimental setup has allowed us to study 30,620 faults, across all modules of the OpenSPARC T1 processor core design. Figure 2 shows the outcome of these experiments for each module, where the total fault injections for the module are normalized to 100%. Overall we observe that 59.9% of the faults are masked, 29.1% are detected, and only 0.94% result in SDCs (conservatively). The remaining 10.1% are in the timeout or other categories. We analyze our results in detail below:

Masking: We observe a high masking rate of 59.9% on average for permanent faults across all the modules. This is higher than the masking rate observed in previous microarchitecture-level permanent fault injection results [3] (16%) or the gate-level results for the three modules simulated with SWATSim (30% to 40%). We believe that the masking rate is high because: (1) The OpenSPARC core was originally designed for 4 hardware threads. However, our experiments used the one threaded version of the core. Although only one thread is functional, the pruning in the design was not complete, leaving unutilized hardware components needed for multithreaded execution. Such hardware, when injected with faults, can raise the masking rate. (2) Some modules such as MUL, FFU, and TLU contain paths that are not significantly exercised by our applications. For example, our applications do not contain streaming or floating point instructions that use all MUL and FFU features. The masking rate for the TLU may also have been impacted by the little exception handling required for our SPEC applications. Also, roughly 1/6th of TLU fault injections were in the performance counter logic and were all masked. (3)

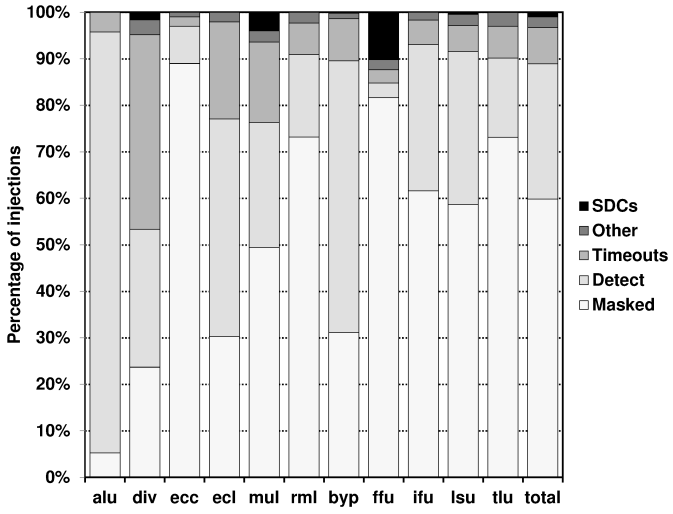

Fig. 2. Breakdown of fault injections by unit. The x-axis shows the unit under study, and the y-axis shows the percentage of experiments that fell into each category.

There is natural circuit and application level masking where the faulty path is exercised but the faulty value does not affect the application output; e.g., our detailed results (not shown here) show that there is more masking for stuck-at-0 than stuck-at-1 faults.

Detections: The overall detection rate is 29.07%. In Table III we show the overall detection rate for each of our detectors.

Of the Hardware Stalls, roughly 79% are due to fault injections in the LSU control logic. We also found that Hardware Stalls and Firmware Checks are the only detectors invoked for FFU faults. Overall, the OpenSPARC platform sees a larger variety of detectors invoked relative to previous SWAT simulations.

Timeouts: A significant fraction of the fault injection experiments (7.8%) run much longer than the fault-free execution. In previous work, SWAT developed heuristics to detect software hangs and currently we are investigating whether such detectors can be used to convert the faults in the timeout category into detections.

Others: This category only affects 2.4% of our experiments. It may be the case that a majority of these are caused due to the experimental methodology or latent faults in the OS. We need to further investigate these cases.

SDCs: Our experiments so far have yielded an overall SDC rate of 0.94%. Interestingly, we found that ALU, ECC, ECL, RML, and IFU produced no SDCs. We also noticed that BYP, LSU, and TLU have an SDC rate of $< 1\%$. Only DIV, MUL, and FFU have an SDC rate of over 1%, with FFU having the highest at 10.2%. Thus, the vast majority of the SDCs are concentrated in a few units, which should be the focus of any additional resiliency techniques.

Furthermore, after closer examination of these SDCs, we found 16.6% of them had error messages within their outputs. For example, some cases occurred due to benchmark error consistency checking. Since these produce error messages, it should be possible for a user or Application-level detection mechanism to detect the data corruption.

TABLE III
DETECTION BREAKDOWN

Kernel Panics	Fatal Traps	Firmware Checks	Hypervisor Crashes	Abnormal Exits	Hardware Stalls
31.5%	25.7%	10.8%	9.9%	5.8%	16.2%

IV. CONCLUSIONS AND FUTURE WORK

This paper tested the effectiveness of low-cost fault detectors as in SWAT on an industrial-strength microprocessor core with an extensive number of gate-level permanent fault injections Our gate-level injections are across the entire processor – previous studies of permanent faults were limited either to microarchitecturally visible structures or could explore only a few modules. Our fault injections were accomplished with the support of the CrashTest resiliency framework, a tool that can automatically insert faults in the gate-level model of the design under test. Our results validate the previous promise of lightweight detection techniques, but also exposed some interesting phenomena, including the concentration of SDCs in a few modules and variety in the detectors invoked in a real system.

We injected a total of 30,620 stuck-at faults throughout the major hardware modules of the OpenSPARC core. The current set of SWAT detectors were able to detect 72.4% of unmasked faults, and many of the remaining undetected cases may be application or OS hangs. Overall, only 0.94% of the experiments led to silent data corruptions.

We would like to extend this work in several directions. First, we would like to evaluate more fault models with our higher coverage of the OpenSPARC hardware. Second, we would like to implement SWAT hang detectors to better evaluate and understand our timeout cases. Third, we want to further evaluate the effectiveness of the SWAT detectors and measure their latency in detecting hardware faults. Finally, we would like to extend this work to a dual core system to evaluate a full detection, diagnosis, and recovery scheme for SWAT.

V. ACKNOWLEDGMENTS

The work at the University of Michigan was developed with partial support from the National Science Foundation and the Gigascale Systems Research Center.

The University of Illinois work is supported in part by the Gigascale Systems Research Center (funded under FCRP, an SRC program), the National Science Foundation under Grant CCF 0811693, and a Computing Innovations fellowship.

REFERENCES

[1] S. Borkar, "Designing Reliable Systems from Unreliable Components: The Challenges of Transistor Variability and Degradation," *IEEE Micro*, vol. 25, no. 6, 2005.

[2] M. Dimitrov and H. Zhou, "Unified Architectural Support for Soft-Error Protection or Software Bug Detection," in *International Conference on Parallel Archtectures and Compilation Techniques*, 2007.

[3] M. Li, P. Ramachandran, S. Sahoo, S. Adve, V. Adve, and Y. Zhou, "Understanding the Propagation of Hard Errors to Software and Implications for Resilient Systems Design," in *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2008.

[4] G. Lyle, S. Cheny, K. Pattabiraman, Z. Kalbarczyk, and R. Iyer, "An End-to-end Approach for the Automatic Derivation of Application-Aware Error Detectors," in *International Conference on Dependable Systems and Networks*, 2009.

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

[6] K. Pattabiraman, G. Saggese, D. Chen, Z. Kalbarczyk, and R. Iyer, "Dynamic Derivation of Application-Specific Error Detectors and their Implementation in Hardware," in *European Dependable Computing Conference*, 2006.

[7] P. Racunas, K. Constantinides, S. Manne, and S. S. Mukherjee, "Perturbation-based Fault Screening," in *International Symposium on High Performance Computer Architecture*, 2007.

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

[9] M. Li, P. Ramachandran, R. U. Karpuzcu, S. Hari, and S. Adve, "Accurate Microarchitecture-Level Fault Modeling for Studying Hardware Faults," in *International Symposium on High Performance Computer Architecture*, 2009.

[10] S. Hari, M.-L. Li, P. Ramachandran, B. Choi, and S. V. Adve, "mSWAT: Low-cost Hardware Fault Detection and Diagnosis for Multicore Systems," in *International Symposium on Microarchitecture*, 2009.

[11] A. Pellegrini, K. Constantinides, D. Zhang, S. Sudhakar, V. Bertacco, and T. M. Austin, "CrashTest: A Fast High-Fidelity FPGA-based Resiliency Analysis Framework," in *ICCD*, 2008.

[12] P. Civera, L. Macchiarulo, M. Rebaudengo, M. S. Reorda, and M. Violante, "FPGA-based fault injection techniques for fast evaluation of fault tolerance in VLSI circuits," *Lecture Notes in Computer Science*, vol. 2147, 2001.

[13] C. López-Ongil, M. García-Valderas, M. Portela-García, and L. Entrena-Arrontes, "An autonomous FPGA-based emulation system for fast fault tolerant evaluation," in *FPL*, 2005.

[14] P. Ramachandran, P. Kudva, J. W. Kellington, J. Schumann, and P. Sanda, "Statistical Fault Injection," in *International Conference on Dependable Systems and Networks*, 2008.

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

[16] D. Sorin, M. M. K. Martin, M. D. Hill, and D. A. Wood, "SafetyNet: Improving the Availability of Shared Memory Multiprocessors with Global Checkpoint/Recovery," in *International Symposium on Computer Architecture*, 2002.

[17] M. Prvulovic, Z. Zhang, and J. Torrellas, "ReVive: Cost-Effective Arch Support for Rollback Recovery in Shared-Mem Multiprocessors," in *International Symposium on Computer Architecture*, 2002.

[18] A. J. KleinOsowski and D. J. Lilja, "Minnespec: A new spec benchmark workload for simulation-based computer architecture research," *IEEE Comput. Archit. Lett.*, vol. 1, 2002.