Relyzer: Application Resiliency Analyzer for Transient Faults

Siva Kumar Sastry Hari[1,2], Helia Naeimi[2], Pradeep Ramachandran[1], and Sarita V. Adve[1]

[1]Department of Computer Science, University of Illinois at Urbana Champaign, swat@uiuc.edu
[2]Intel Labs, Intel Corporation, helia.naeimi@intel.com

***Abstract*—Future microprocessors need low-cost reliability solutions to enable reliable operations in the presence of failure-prone devices. The state-of-the-art reliability solutions detect the presence of hardware faults by deploying low-cost software-level symptom monitors. Recently researchers have shown that these detection mechanisms provide high fault coverage with only few faults being undetected. There is a risk that these undetected faults can result in silent data corruptions or SDCs. The SDC rates demonstrated by the state-of-the-art symptom detection mechanisms have been an impressive <0.5% for permanent and transient hardware faults in all hardware units studied except the data-centric FPU. However, a thorough and accurate analysis is needed to evaluate the SDC rate to make these techniques practically viable.**

This paper presents Relyzer, an approach that analyzes the fault-free execution of applications and performs smart selective fault injection experiments, as opposed to random fault injections. Relyzer can thus provide a tight bound on the SDC rate. Relyzer first lists all the architecture level hardware faults that can possibly affect an application. It then employs a set of novel fault pruning techniques to eliminate a large fraction of them by predicting their outcomes and showing them equivalent to others. The hardware faults that remain after the pruning phase are the only ones that need thorough fault injection experiments. Our results show that Relyzer is capable of pruning about 99.9979% of hardware faults for the workloads that we studied. Some of our pruning techniques are heuristic based and validating them is a part of our ongoing work.

***Index Terms*—Silent Data Corruption, Transient Faults, Dynamic Program Analysis, Architecture**

I. Introduction

As process technology scales, hardware reliability is becoming a major challenge. The likelihood of future hardware failing in the field is increasing due to various reasons such as wear-out, transient errors, and design bugs [1]. Future systems must handle such failures through in-field fault detection, diagnosis, repair, and recovery to guarantee continuous reliable operations. Fault detection mechanisms form a crucial part in devising such low-cost reliability solutions. Traditional systems use heavy amounts of redundancy (in space or time) to detect faults. Owing to prohibitive cost, such detection mechanisms are no longer accepted for modern commodity systems. An effective detection technique should minimize the faults that escape detection and corrupt application outputs; such faults are commonly known as Silent Data Corruptions or SDCs.

Software-level symptom based fault detection mechanisms [2, 3, 4, 5, 6, 7, 8, 9] have emerged as viable low-cost alternatives. These detect only those hardware faults that corrupt software execution by monitoring anomalous software executions. SWAT (SoftWare Anomaly Treatment) [4, 10, 11, 12], a state-of-the-art reliability solution, is one such example. Despite the simplicity of its detectors, SWAT achieves an impressive SDC rate of <0.5% across several (compute-intensive, media, and distributed client-server) workloads for permanent and transient faults in all hardware units studied except the data-centric FPU. The effectiveness of SWAT has been demonstrated through statistical methods. Statistical techniques randomly select a small representative sample of hardware faults (typically in the order of 10s of thousands) out of all hardware faults in the system (typically in the order of 100s of trillions). While these methods provide statistical guarantees on coverage and SDC rates, certain faults that may be important to the application may be sampled out. Hence, an accurate measurement of the SDC rate is required to make SWAT-like symptom detectors a practically viable solution.

An ideal alternative is, therefore, to perform exhaustive fault injections to measure accurate SDC rate. This can also list all possible SDC-causing hardware faults and fault propagation patterns which can be vital for eliminating them in a cost-effective manner. However, this is impractical as the number of such faults are in the order of trillions. Therefore performing a systematic and thorough analysis of all the hardware faults while running an application is a major challenge.

This paper presents Relyzer, a technique that is capable of systematically analyzing all architecture level transient hardware faults that can affect an application. Relyzer first attempts to predict the outcome (detection, masking, or SDC) of a transient fault. If the outcome of the fault is predictable then it need not be studied through a fault injection experiment. Relyzer then attempts to categorize faults that behave similarly in to equivalent classes and select one representative for fault injection. This further reduces the number of faults that require fault injection study. Overall, Relyzer bins all the hardware faults affecting an application into the following three categories:

- *Predictable faults*: Using static and dynamic analysis,

The authors acknowledge the support of the Gigascale Systems Research Center, one of six research centers funded under the Focus Center Research Program (FCRP), a Semiconductor Research Corporation entity. This work also supported in part by the the National Science Foundation under Grant CCF 0811693. Pradeep Ramachandran was supported by an Intel PhD fellowship and an IBM PhD scholarship.

Relyzer attempts to predict the outcome of a hardware fault. If Relyzer is able to predict the outcome then we do not inject the fault to study it.

- *Equivalent faults*: Hardware faults that are expected to show similar behavior fall under this category. We group all equivalent hardware faults into one bin and then select just one representative. We use heuristics to show the equivalence between different faults. Studying the behavior of such representative faults is sufficient to determine the outcome of all faults in an equivalence groups.
- *Remaining faults*: These are the unique hardware faults with outcomes that Relyzer cannot determine. These faults (faults where outcome cannot be predicted and faults that represent the equivalent groups) need fault injection experiments to determine their outcomes.

We present a series of fault pruning techniques to keep the set of hardware faults that fall in the *remaining faults* category small such that performing fault injections on all of them would be feasible. Overall, Relyzer prunes out the set of hardware faults such that studying a small number of them is sufficient to determine the outcomes of all the faults affecting an application. Our results show that Relyzer prunes over 99.99% of 1.16 trillion hardware faults across four applications and only 26 million remaining faults require thorough fault injection experiments.

The validation of the heuristics based fault pruning techniques, used by Relyzer, is an ongoing work. Assuming that the pruning performed by Relyzer is sound, it would be feasible to exhaustively inject faults from the *remaining* category to obtain a tightly bound SDC rate. Relyzer preserves the ability to list all SDC-causing hardware faults. This ability can potentially drive the development of low-cost fault detectors (hardware or software). Relyzer also has the ability to identify portions of an application that are SDC prone and provide feedback to the programmer to assist him/her to further improve the resiliency of the application.

II. Related Work

SWAT [4, 10, 11, 12] is a low-cost comprehensive reliability solution for in-core faults. It detects hardware faults by monitoring anomalous software executions through low-cost symptom detectors. The SDC rates demonstrated by the SWAT detectors have been an impressive <0.5% of injected permanent and transient hardware faults in all hardware units studied except the data-centric FPU. SWAT evaluates its detection mechanism using statistical fault injections on a micro-architecture level model [4] and on a more accurate gate level fault model [13]. SWAT, however, provides only empirical evidence for its low SDC rate. Relyzer, on the other hand, aims to analyze all faults in an application and provide a tightly bound SDC rate.

To accurately measure SDC rate, techniques that analyze all faults are required and SymPLFIED [14] has such capability. The focus of SymPLFIED is to highlight the faults that result in detection. SymPLFIED uses symbolic executions to abstract state of erroneous values in the program and model checking to find all errors that evade detection. SymPLFIED was applied on only few small Siemens benchmarks. It is not clear whether this search based approach will scale to larger applications.

Performing fault injection experiments on all hardware faults is time consuming and impractical. Benso et al. [15], therefore, proposed a solution that performs run-time analysis of the application variables to obtain the criticality behavior of every variable. It first develops an analytical model to compute the criticality of a variable. It then performs a series of fault injections to initialize the model parameters. Finally, it employs the model with initialized parameters on other applications. The results demonstrated that this solution has low inaccuracies in predicting which variables are more likely to cause SDCs and are thus critical to protect. However, the results were shown on small applications with few variables. Relyzer, on the other hand, aims to provide high confidence measurements even for complex and large applications.

Sidharan et al. [16] performed a study of hardware fault behavior at the application level. It quantifies the reliability behavior of an application using a metric called Program Vulnerability Factor (PVF). PVF is a microarchitecture independent method to quantify architectural fault masking inherent to a program. PVF provides insights into the application level reliability by evaluating different implementations of the same application. However, PVF focuses on identifying only those faults that are masked by the application. It does not attempt to distinguish faults that lead to SDCs from the ones that result in detection. However, with a SWAT-like system in place, it becomes crucial to distinguish faults that result in detection from those that result in SDCs because the system can now be recovered from faults that result in detection. Hence, Relyzer focuses on distinguishing SDCs from detections. It is unclear whether PVF can make this distinction.

III. Relyzer: Overview

Relyzer systematically analyzes all hardware faults in an application and categorizes them as *predictable*, *equivalent*, and *remaining faults* (Fig 1). The outcomes of *predictable faults* can be determined with simple observations and analysis. These faults do not need fault injection experiments. *Equivalent faults* are those that are shown to have similar behavior. All the equivalent faults are grouped together and only one representative is picked for fault injection. The hardware faults that are not categorized as predictable or equivalent fall under the *remaining faults* category.

We developed a set of novel pruning techniques to identify hardware faults that fall under the *predictable* and *equivalent* categories. For identifying predictable faults, we used two pruning techniques, namely bounding addresses and bounding branch targets. These techniques aim to identify hardware faults that result in catastrophic failures upon activation (e.g., segmentation faults, application aborts, kernel panic, etc.). For identifying faults that fall under the equivalent category, we developed two types of pruning techniques. The first type uses def-use analysis and constant propagation to show how two faults can have same outcomes. The second type uses heuristics based algorithms on fault-free executions to identify

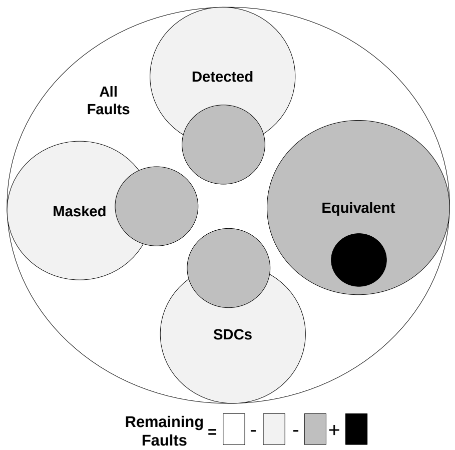

Fig. 1. Overview: Sections marked detected, masked, and SDCs represent the faults that have predictable outcomes. Sections marked equivalent are the faults that are shown to be equivalent to others. The black section is the representative set of unique faults that require fault injections to determine the outcome. Faults that fall under the *remaining faults* category are shown at the bottom of the figure.

computation patterns that can potentially behave similarly under the influence of faults. Lastly, we added one more pruning technique that uses statistical methods to evaluate and prune faults in the branch instructions.

In this study we consider all ISA visible architecture level faults. Since the focus is on transient faults, we consider single bit-flips in the destination and source operands (in both register and immediate values). If the operand is used as an address then we also consider faults in the computed virtual address and the memory value accessed by the instruction.

IV. Fault Pruning Techniques

A. Pruning Faults with Predictable Outcomes

Bounding addresses: Hardware faults can make applications access memory locations that fall out of the range of the allocated address space. Such accesses are likely to be detected by operating systems and result in detectable symptoms (for example, segmentation faults, application aborts, and kernel panic). In fact, SWAT employs out-of-bounds and app-abort detectors specifically to detect such scenarios within recoverable latencies. Hence we do not need injection experiments to identify the outcome of the faults in the of addresses that would make them access unallocated addresses. We can directly prune them from the set of hardware faults.

We determine the range of valid addresses, for both stack and heap, by studying the dynamic memory profile of the application. To keep our implementation simple, we monitor global and heap addresses together. This also eliminates the problem of distinguishing them from each other during runtime. Once we identify the range of the valid addresses, we prune faults in the bits of any memory access that would allow them to access an invalid address.

Bounding branch targets: A fault that causes the control to jump to a location that is not in the application may result in a detectable symptom (e.g. application abort, segmentation fault, etc.). SWAT's app-abort and fatal-trap detectors can detect these scenarios. We identify the addresses that constitute the valid targets by analyzing the text section of an application. Typically, the text section is small (under million instructions, i.e., under 32 bits) and hence a large fraction (over 50% on 64-bit machines) of faults in the branch targets can be predicted as detected and pruned by this technique alone.

B. Pruning Equivalent Faults

Def-use analysis: A register definition is created whenever a register is used as a destination register in an instruction. Faults in the use-chain of this definition, formed by those instructions that use this register as a source operand, have similar behavior to faults in this definition. Therefore, we prune out faults in the use-chain and retain faults only in the definition. Since all destinations have at least one use in the application, we expect this technique to provide high pruning. If a definition has no use then the instruction is dead and we prune all faults in this instruction.

Constant propagation: This pruning technique observes the fault propagation pattern in the instructions that operate on constants. Currently this technique is limited to only those logical operations that preserve the propagation of a single-bit fault from the source operands to a destination operand. It then prunes the faults in the source operands and retains only the faults in the destination operands.

The techniques discussed so far do not use any approximations to prune the faults. Our results, interestingly, show that over 47% of the application-level hardware transient faults are pruned by these techniques across all our applications (we discuss them later in the evaluation section).

We observed that most of the remaining transient faults come from three main sections of a basic block - instructions that lead to a store, instructions that lead to a function call, and instructions that lead to a compare instruction. In the following sections we focus on developing techniques to prune faults in these three code sections.

Dynamic store-load analysis: This pruning technique explains how a fault in two dynamic store instructions can be shown equivalent (figure 2). We believe that faults in different store instructions can be shown equivalent if the store instructions have similar store effects. We measure the store effect by using a heuristic. Assuming that the two store instructions write in different addresses (e.g., Store 1 and Store 2 in figure 2), we first check whether the number of loads for these addresses are same. If this is the case, then we check the locations of these loads in the static control flow graph, by looking at the program counter (e.g., comparing PC-L1a to PC-L2a and PC-L1b to PC-L2b in figure 2). If these also match, then we have a high degree of confidence that the two store instructions have similar effects. To gain more confidence we also look at the dynamic control flow at each of the load instructions. In our simulation setup, we observe ten previous memory instructions to obtain the dynamic control flow at each of these locations. If the dynamic control flow also matches, we conclude that the two dynamic store instructions have similar store effects and we prune faults from one of them.

Call-site analysis: Whenever a function is called, the function parameters are transferred to the function body though

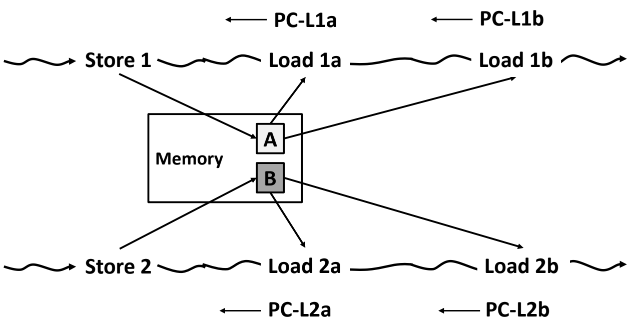

Fig. 2. Example explaining dynamic store-load analysis: Store 1 and Store 2 are two store instructions writing to different addresses. Load 1a with program counter PC-L1a and Load 1b with program counter PC-L1b are two load instruction reading the value from address A. Similarly Load 2a and Load 2b are two loads from address B at locations PC-L2a and PC-L2b respectively. If PC-L1a is equal to PC-L2a and PC-L1b is equal to PC-L2b then we have some confidence that the two stores (store 1 and store 2) have similar effects.

registers. These register are later stored onto different stack locations. Similarly the return value is passed from the function body to the call-site though a register. This register is later stored onto a memory location (stack or heap). Hence call-site forms a special case of the store instruction and we can utilize the dynamic store-load analysis discussed above to prune faults in this code section.

Dynamic compare analysis: Compare instructions are usually followed by branch instructions. A fault in the compare instruction and the instructions that lead to it often affects only the direction of the branch. Hence knowing the effect of an incorrect branch and the number of faults that make the branch take the incorrect direction, we can estimate the outcomes of all the faults in the code section that leads to a compare instruction. We obtain the effect of the wrong direction by selecting only one fault among all the faults that make the branch take the incorrect direction. We obtain the distribution of the faults that lead to a correct or an incorrect direction by first identifying a dynamic instance of the compare instruction and selecting the code section that leads to this compare instruction. We then inject all possible faults in this dynamic instance of the code section.

C. Statistical Pruning

These pruning techniques are employed only when the pruning from previous techniques is not sufficient. We currently use this approach only for the branch instruction.

Remaining branch targets: Faults in the branch targets that are not pruned by the *bounding branch targets* technique fall under this category. For every static conditional branch instruction, we select a statistical sample of dynamic instances and inject faults in all the bits in the branch target.

Remaining branch direction: The faults that fall in this category are the ones that are remaining from after applying *dynamic compare analysis*. These are the one-bit faults that are used to measure the affect of an incorrect branch direction. We select a statistical sample of dynamic instances for every conditional branch instruction to inject the one-bit direction fault.

TABLE I
APPLICATIONS

Application	Description	Number of Instructions	Number of Faults
LU (SPLASH-2)	Factors a matrix into the product of lower & upper triangular matrix	2.1 Billion	310 Billion
FFT (SPLASH-2)	1D Fast Fourier Transformations	7.1 Billion	111 Billion
Blackscholes (PARSEC)	Calculates prices of options with Black-Scholes partial differential equation	1.7 Billion	214 Billion
Swaptions (PARSEC)	Computes prices of a portfolio of swaptions using Monte Carlo simulations	2.7 Billion	534 Billion

We select the size of the sample of dynamic instances for every static branch instruction such that we have high confidence (99%) and low error (5%) in the results.

We plan to devise a heuristic based fault pruning technique even for the branch instructions and eliminate the statistical sampling approach altogether.

V. EVALUATION

We implemented the above described pruning techniques on the SPARC V9 [17] architecture. As mentioned before, we study all transient faults in the ISA visible architecture states. We consider transient faults (bit-flips) in the destination and source operands of every instruction. If the operand is used as an address then we also consider faults in the computed virtual address and the memory value accessed by the instruction.

We evaluate the developed pruning techniques on four applications – two each from SPLASH-2 [18] and PARSEC [19] benchmark suits. A brief description of the applications, the length of each application (based on the inputs selected), and the number of faults prior to applying any pruning is shown in Table I.

We required both static and dynamic analysis of the application to implement the pruning techniques. The static analyzer traverses the application and creates the set of all hardware faults. It then applies static pruning techniques and computes the pruned fault set. The dynamic analyzer profiles the branches and the store-load patterns to implement the designed pruning techniques. We use Virtutech simics [20] to implement these dynamic profilers. We use the information from both static and dynamic analyses to generate the final pruned fault set.

The size of the initial set of hardware faults across all applications, before applying Relyzer, was approximately 1.16 trillion. It is practically impossible to study these faults by performing fault injection experiments. After applying all our developed pruning techniques Relyzer was able to reduce the set of hardware faults such that it is feasible to perform fault injection experiments on all the remaining faults. Relyzer reduced the set of hardware faults to under 26 million, which corresponds only 0.0021% of the inital set of hardware faults.

Table II shows the breakdown of pruning obtained by different pruning techniques. As shown in table II, approximately 3.8% of the faults were pruned by predicting the outcomes. Non-approximation based techniques that show equivalence

TABLE II
THE BREAKDOWN OF THE PRUNING OBTAINED FROM EACH OF THE TECHNIQUES.

Class	Pruning technique	Does it use heuristics?	Pruned
Predictable	Bounding addresses	No	2.8690%
	Bounding branch targets	No	0.9404%
Equivalent	Def-use analysis	No	37.5772%
	Constant propagation	No	5.9492%
	Dynamic store-load analysis	Yes	43.7386%
	Call-site analysis	Yes	2.6213%
	Dynamic compare analysis	Yes	5.8568%
Statistical	Remaining branch targets and direction	N.A.	0.4453%
Total			99.9979%
Remaining			0.0021%

between different faults provided about 43.2% of pruning and the heuristic based techniques provided about 48.6% pruning. This makes the pruning obtained by showing the equivalence between different hardware faults to about 91.8%. Lastly, the statistical sampling method for branch targets and direction provided about 0.45% of pruning. Overall the pruning of 99.9979% shows how effectiveness Relyzer can be in pruning hardware faults.

In ongoing work, we are validating the pruning performed by the dynamic store-load analysis. We also plan to validate the remaining heuristic based pruning techniques in near future. Once we validate all our pruning techniques, we will focus on obtaining the effective SDC rate.

VI. Conclusions and Future Work

Hardware reliability has become a major challenge in the late CMOS era. Hence low-cost and effective fault detection mechanisms are needed. Symptom based fault detection mechanisms have emerged as low-cost alternatives. SWAT, a state-of-the-art reliability solution, employs such low-cost detectors. It statistically demonstrated that this detection mechanism produces only 0.5% of hardware faults as unacceptable Silent Data Corruptions or SDCs. However, the SDC rate needs to be accurately measured and analyzed to employ such mechanisms in commodity systems. Accurately measuring and reducing the SDC rate is vital for any fault detection mechanism.

This paper presents Relyzer, a group of novel fault pruning techniques, that can analyze all hardware faults and significantly reduce the number of faults to study. Relyzer predicts the outcome of several faults, eliminating the need for thorough fault injection experiments. Relyzer exploits the fact that several faults exhibit similar behavior to a small set of hardware faults to further prune faults. Relyzer employs a series of static and dynamic pruning techniques to reduce the set of hardware faults from 1.16 trillion to under 26 million, a pruning of 99.9979%. It is feasible to exhaustively perform fault injection experiments on these pruned hardware faults to measure the SDC rates with tight bounds.

We recently finished the implementation of the fault injection infrastructure. We also started validating the heuristics based pruning techniques used in this paper. In near future, we plan to perform fault injection experiments on the remaining faults and relate them to all the equalized faults in order to calculate the effective SDC rate.

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] O. Goloubeva *et al.*, “Soft-Error Detection Using Control Flow Assertions,” in *DFT*, 2003.

[4] 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.

[5] 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.

[6] A. Meixner *et al.*, “Argus: Low-Cost, Comprehensive Error Detection in Simple Cores,” in *International Symposium on Microarchitecture*, 2007.

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

[8] P. Racunas *et al.*, “Perturbation-based Fault Screening,” in *International Symposium on High Performance Computer Architecture*, 2007.

[9] 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.

[10] M. Li, P. Ramachandran, S. Sahoo, S. Adve, V. Adve, and Y. Zhou, “Trace-Based Microarchitecture-Level Diagnosis of Permanent Hardware Faults,” in *International Conference on Dependable Systems and Networks*, 2008.

[11] S. Sahoo *et al.*, “Using Likely Program Invariants to Detect Hardware Errors,” in *International Conference on Dependable Systems and Networks*, 2008.

[12] S. K. S. Hari, M. Li, P. Ramachandran, B. Choi, and S. V. Adve, “Low-cost Hardware Fault Detection and Diagnosis for Multicore Systems,” in *International Symposium on Microarchitecture*, 2009.

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

[14] K. Pattabiraman, N. Nakka, Z. Kalbarczyk, and R. Iyer, “Symplfied: Symbolic program-level fault injection and error detection framework,” in *International Conference on Dependable Systems and Networks*, 2008.

[15] A. Benso, S. Di Carlo, G. Di Natale, P. Prinetto, and L. Taghaferri, “Data criticality estimation in software applications,” in *International Test Conference*, 2003.

[16] V. Sridharan and D. R. Kaeli, “Eliminating microarchitectural dependency from architectural vulnerability,” in *International Symposium on High Performance Computer Architecture*, 2009.

[17] D. L. Weaver and T. Germond, eds., *The SPARC Arch. Manual.* Prentice Hall, 1994. Version 9.

[18] S. C. Woo *et al.*, “The SPLASH-2 Programs: Characterization and Methodological Considerations,” in *Proc. 22nd Intl. Symp. on Comp. Arch.*, 1995.

[19] C. Bienia and K. Li, “PARSEC 2.0: A New Benchmark Suite for Chip-Multiprocessors,” in *Proc. of 5th Workshop on Modeling, Benchmarking and Simulation*, 2009.

[20] Virtutech, “Simics Full System Simulator.” Website, 2006. http://www.simics.net.