Understanding Why Symptom Detectors Work by Studying Data-Only Application Values

Pradeep Ramachandran†, Siva Kumar Sastry Hari†, Sarita V. Adve†, and Helia Naeimi‡
†Department of Computer Science, University of Illinois at Urbana-Champaign
swat@cs.uiuc.edu
‡Intel Labs, Intel Corporation
helia.naeimi@intel.com

***Abstract*—Failures from unreliable hardware are posing a serious threat to system reliability. Symptom detectors that monitor anomalous software execution to detect such failures are emerging as a viable low-cost detection scheme for future systems. Since these detectors do not provide perfect resiliency guarantees, they strive towards reducing instances when faults escape detection and affect application output; such faults are commonly referred to as Silent Data Corruptions (SDCs). Previous work on symptom detection has demonstrated low SDC rates through empirical fault injection experiments.**

This paper, for the first time, presents an intuitive reasoning behind why symptom detectors achieve such low SDC rates. The key insight is that faults that propagate to control operations and/or memory addresses are easier to detect than those that affect pure data computations. We show that less than 5% of the application values fall into this category, making symptom detection effective. Further, we demonstrate that faults in such values yield high SDC rates, warranting special attention. Finally, we explore prior techniques to identify critical data-only values that may lead to SDCs and find that while they show large promise to identify the critical values, the detectors placed at those locations need to be carefully designed in order to keep the false positives rates under check.

I. Introduction

Failures from unreliable hardware are posing a serious threat to system reliability in the late CMOS era [2]. The reasons for such failures include, but are not limited to, manufacturing defects, early-life failures, and in-field failures. With such failures threatening future commodity systems, traditional solutions that employ heavy redundancy are too expensive, warranting alternate solutions.

Symptom detection is emerging as a promising low-cost alternative to such traditional solutions [3], [6], [8], [11], [13], [15]. These detectors allow the hardware fault propagate to the software and employ low-cost techniques that monitor symptoms of anomalous software behavior for fault detection.

Symptom detectors do not, however, guarantee perfect detection of all hardware faults. While faults masked by the lower levels are safely ignored, some faults escape detection and affect application outputs, resulting in SDCs. These detectors ride the trade-off curve between overheads (in performance, area, power) and resiliency, aiming to provide acceptably low SDC rates at minimal overheads. The SWAT system represents the state-of-the-art in symptom detection and has demonstrated that simple low-cost detectors result in low SDC rates of under 0.5% for permanent and transient hardware faults in all hardware units studied, except the data-centric FPU, with a variety of single-threaded and multi-threaded workloads on multicore systems [6], [7], [8], [13].

Despite the promise of low SDC rates at low overheads, the evidence for the success of such symptom detectors has been largely empirical. An analytical understanding of these hard-to-detect SDCs has not been built this far and is important for symptom detection for the following reasons. First, the SDC rates, although low, may not be low enough for certain application classes such as mission critical systems, and financial applications. Understanding these SDCs may yield better detectors and lower the SDC rates even further. Second, there has been much work on devising low-cost software-level detectors to detect hardware faults in software for applications that warrant lower SDC rates [13]. Understanding these hard-to-detect faults may help to identify the vulnerable parts of the software that need may be protected through such mechanisms. Third, the statistical fault injectors used to evaluate the detectors can be guided to inject these hard-to-detect faults and stress-test the detectors.

There are two axes along which the characteristics of these SDCs need to be studied – an application-centric axes and a hardware-centric axes. An application-centric characterization would help identify application values in which faults may be harder to detect, enabling application hardening against hardware faults. A hardware-centric characterization would help in identifying those hardware structures in which faults may be harder to detect and can be used to choose which hardware structures to protect with mechanisms like ECC.

This paper takes an application-centric view towards these hard-to-detect SDCs and studies how faults propagate through the application (the hardware-centric evaluation is left to future work). It answers the following questions about symptom detectors – Why are symptom detectors, like SWAT, effective in detecting hardware faults? Where do they fall short? What values should we target for future detectors? The key

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 Grants NSF CCF 0541383, CNS 0720743, and CCF 0811693, and an OpenSPARC Center of Excellence at Illinois supported by Sun Microsystems. Pradeep Ramachandran was supported by an Intel PhD fellowship and an IBM PhD scholarship.

insight used is that symptom detectors, like SWAT, rely on some form of deviation in control flow or memory addresses for successful detection. Faults in values that do not affect such operations, classified as *data-only values*, are therefore vulnerable under symptom detection. In this paper, we focus on these values and show that:

1) Only a small number of application values ($< 5\%$) fall under the data-only value category, explaining the low SDC rates from symptom detectors used in SWAT.
2) Faults in data-only values are harder to detect than faults in random values, as a higher fraction result in SDCs or are unrecoverable as they are detected at long latencies. For transient faults injected into the SPEC workloads, we find that approximately 4% of the faults are hard-to-detect when the injected values are chosen at random, while when the values are picked from those that are known to be data-only, 13% of the faults are hard-to-detect. Data-only values thus need additional detection mechanisms to identify such hard-to-detect faults.
3) Previous metrics to identify vulnerable application values, when applied to data-only values are successful in identifying the critical values. We demonstrate that the fanout metric developed in previous work [10] helps select locations for fault detectors that cover nearly all the critical data-only values that cause hard-to-detect faults. The detectors themselves, however, need to be carefully designed as they may incur a high rate of false positives if they do not consider dynamic values.

Thus, this paper, for the first time, provides an explanation for why symptom detection strategies such as SWAT work as well as they do (relatively rare occurrence of data-only values) and shows that there is a certain class of values that need additional focus to improve the SDC rates (data-only values).

II. Related Work

Symptom detectors have received much attention of late to handle software bugs [4], transient hardware faults [11], [10], [15], and permanent hardware faults [3], [6], [8], [13]. In this paper, we regard SWAT as exemplifying this approach and use its detectors, fatal traps, hangs, application aborts, kernel panics, high OS, and address-out-of-bounds, for fault detection [8]. The iSWAT framework supplemented these detectors with compiler-derived software-level detectors to lower the SDC rate by detecting faults in data values [13]. We do not use these software-level detectors here due to limitations in our infrastructure to instrument the application with such invariants. Additionally, the work presented here provides insights into the types of values need additional protection. These insights may be used to guide where such, and other, software-level detectors may be placed in order to make them most effective.

There is a growing interest in modeling how hardware faults propagate through the application for such symptom detectors [1], [9], [10], [12], [14]. Benso et al developed a software-level analytical model to predict which application variables are critical for SDCs and demonstrated its accuracy with small benchmarks [1]. Other work developed models to depict how hardware faults lead to application crashes [9], [10], [12] and derived hardware-level detectors [9] and application-level detectors [10], [12] from such models. The application-level detectors were chosen based on metrics such as fanout, lifetime, etc. that signify the criticality of the value to propagate the fault. By selectively protecting these variables, they demonstrated that the detection latency and SDCs may be reduced. The PVF metric also studied fault propagation through a program to remove the microarchitecture-dependent components from AVF [14]. The PVF metric predicts when a fault may be masked by the program but does not distinguish faults that are detected from those that are SDCs.

Although the focus of this work is to also study program properties by understanding how faults propagate through an application, it differs from the above works its goal is to reduce SDCs from symptom detectors; the focus of much of the prior work was on fault detection or to identify masked faults. Further, the techniques presented in this work are evaluated with real-world workloads that contain millions of values and execute for billions of instructions (as opposed to prior work that was largely evaluated on small benchmarks that had 100s of variables and ran for under million instructions), making the results more realistic.

III. Application-centric View of SDCs

Symptom detectors have so far demonstrated that they are highly effective in detecting hardware faults. In particular, the SDC rates demonstrated by the state-of-the-art 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. This low SDC rate is despite the detectors monitoring high level symptoms of software misbehavior.

A. The Data-only Values Classification

One intuition behind such low SDC rates is that symptom detectors detect faults in a majority of the values that affect control instructions and memory addresses. However, those values that affect pure data-only computations (henceforth referred to as a *data-only values*) may not result in software-visible symptoms and hence are harder to detect. The iSWAT framework showed that software-level detectors which detected a subset of these values reduced the SDC rates by up to 74%. In this section, we build an intuition behind why such detectors are effective by understanding how faults propagate through the application.

Figure 1 shows the fraction of data-only values in all C/C++ Integer and floating point workloads of the SPEC CPU 2000 benchmark suite. The data is collected from these applications running inside an unmodified OpenSolaris OS within the Simics full-system simulation environment. We use the GEMS timing models to model an UltraSPARC-III-like processor with a full memory hierarchy (64KB IL1, 64KB DL1, and 2MB L2). For each workload, we pick 4 random phases (each 11 million instructions long) during the application execution and track how data values produced in the first million instructions in each phase are used in the next

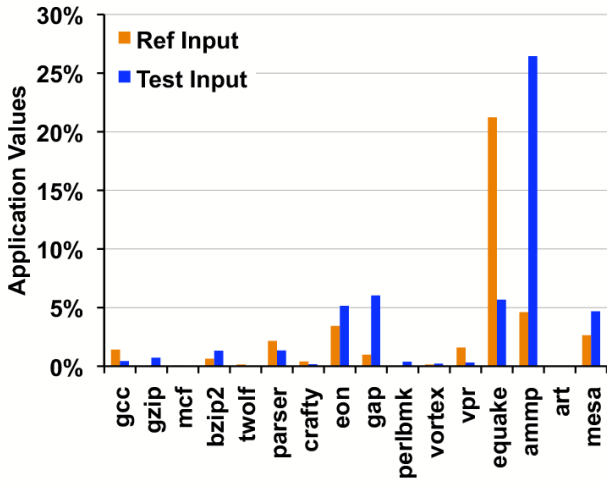


Fig. 1. Data only values in SPEC. The small number of data-only values demonstrates that faults in most values propagate to control instructions and/or memory values, resulting in detectable symptoms. Further, FP workloads have more data-only values making them more vulnerable than integer workloads.

10 million instructions by building the Dynamic Data Flow Graph (DDFG) of the program. If a value does not affect control instructions or memory addresses in this 10 million instruction window, it is classified as a *data-only value*. We do not track the values all the way until application output as the DDFG grows to unmanageable sizes for longer simulation periods; our results are therefore conservative as the set of values we classify as data-only is a super set of the true data-only values.[1] The figure aggregates the number of data-only values across the four phases for each workload. The fraction of values that are data-only are shown for the `ref` and `test` inputs in order to guage the effect of input size on the number of the data-only values.

From this figure we see that the fraction of values that are data-only is fairly low. Nearly all the workloads have $< 5\%$ of values as data-only for both sets of inputs, with the workloads that have FP computations (*eon*, *equake*, *ammp*, *mesa*) having more data-only values than the integer workloads. The three outliers from this observation are the FP workloads *art* for both sets of inputs, *equake* for the `ref` input, and *ammp* for the `test` input. *art* has no data-only values in the measured intervals as it takes several control decisions based on floating point values, unlike other FP workloads. (This workload tries to recognize known objects in a given image, resulting in elaborate FP-value dependent control-flow.) On the flip-side, the fraction of data-only application values is $> 20\%$ for *equake* with the `ref` input set (with 21% data-only values), and *ammp* with the `test` input set (with 26% data-only values). In our chosen phases, these configurations exhibit half an order of magnitude or more floating point operations than the other benchmarks, resulting in many more data-only values. We believe that these outliers are artifacts of the inability of our simulation infrastructure to track data-only values for the entire application, and are not fundamental to these workloads; other randomly chosen phases may not exhibit such abnormal traits.

These results show that faults in a large fraction of the values will eventually affect control decisions and/or memory addresses. Such propagations may result in software-visible symptoms such as branching off to a wrong location or indefinitely looping because of a fault in the control instruction, or out-of-bounds accesses and protection violations from faults in memory addresses. Since symptom detectors, like SWAT, monitor the software for such behaviors, they detect a large fraction of the hardware faults.

B. Detecting Faults in Data-only Values

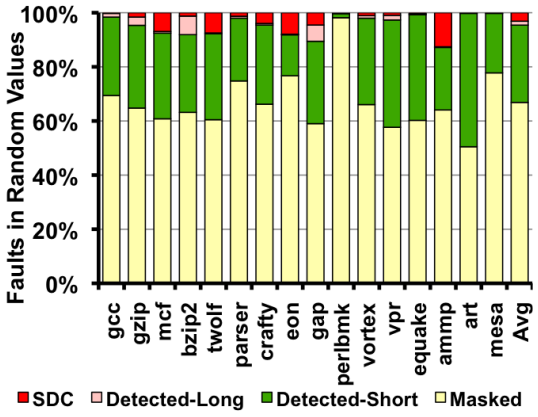
Since faults in such data-only values may not result in visible symptoms, they are expected to be harder to detect than faults in randomly chosen values. Figure 2 compares the outcomes of injecting transient faults into (a) random application values and (b) data-only application values for all 16 C/C++ SPEC workloads with the test input. Each workload is simulated within the afore-mentioned simulation setup and a transient fault is injected (one per run) into a random bit in the destination register of instructions chosen in the first million instructions of each phase. In Figure 2(a), the instruction to inject the fault is chosen at random while for Figure 2(b), faults are injected into those instructions that produce the data-only values shown in Figure 1. For each workload, we inject 4000 faults in random instructions and up to 4000 faults in data-only values. (Some applications have less than 4000 data-only values in our measured window; for these applications, we inject one fault in each instruction that produces a data-only value.)

Once a fault is injected, the system is simulated until either the fault is detected with the SWAT symptom detectors [8] or until the application completes and produces outputs. Faults detected by SWAT in under 100K instructions are classified as *Detected-Short*, and those detected at longer intervals as *Detected-Long*. Undetected faults that do not affect the output of the application are classified as *Masked*, and those that do affect the output as *SDCs*.

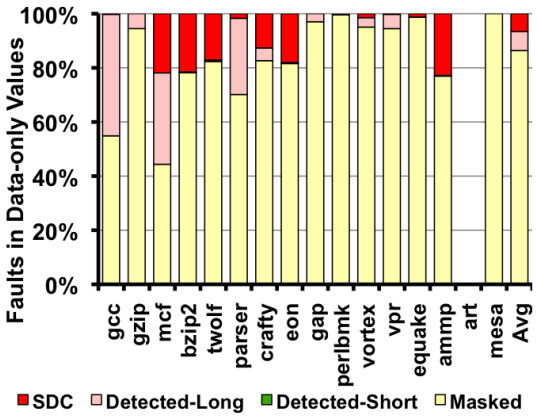
Faults that result in long-latency detections and SDCs are hard-to-detect and require additional support to enable symptom detection and recovery. The results in Figure 2 demonstrate that faults in data-only values tend to produce a higher rate of hard-to-detect faults than faults in random values. On an average, 4.5% of the faults in random values, shown in Figure 2(a), fall under the Detected-Long and SDC categories.[2] On the other hand, a higher 13.4% of the faults in data-only values, shown in Figure 2(b), are hard-to-detect. We also see a similar trends for faults in nearly all the workloads shown. Faults in data-only values may cause detectable symptoms when the values affect control instructions and/or memory operations. Since our infrastructure does not track the propagation until application output, as previously discussed, it categorizes some of these faults as data-only. Nevertheless,

[1]Nevertheless, we found that when this window of propagation was increased to 20 million instructions, the fraction of data-only values reduced by less than 2% for most applications.

[2]Note that the fault model for random faults injected here is different from those used in the previous SWAT papers, which injected transient faults at the microarchitecture-level [8]. This makes our results significantly different from previous results.



(a) Faults in Random Values



(b) Faults in Data-only Values

Fig. 2. Outcome of faults injected into (a) randomly chosen application values, and (b) data-only application values.

all the faults detected in data-only values are detected at long latencies, they are arguably as critical to protect as those that result in SDCs.

Figure 2 shows a couple of other interesting trends in fault masking and fault detection. First, we see that more faults in data-only values are masked than the faults in random values; on the average, 85% of the faults in Figure 2(b) are masked while only 66% of the faults in Figure 2(a) are masked. Faults in random values show lower masking rates as the values affected by the fault may be used for a variety of operations including control-flow and memory addressing. Faults in data-only values, on the other hand, are used purely for data-flow and have a higher chance to be logically masked. Second, we see that over 86% of the unmasked faults in random values are detected in under 100K instructions by the SWAT detectors (*Detected-Short* in Figure 2(a)), demonstrating that the SWAT detectors are highly effective in detecting faults in random application values.

These results show that faults in data-only values are hard-to-detect with existing symptom detectors. Additional support from the application may be required to enable symptom detection for faults in such values.

C. Identifying Critical Data-only Values

Since faults in data-only values are hard to detect, as demonstrated in the previous section, we analyze their data-flow properties in an attempt to predict which faults are more likely to cause hard-to-detect faults than others. In particular, we explore the previously proposed metric of fanout of a value [10] to determine its criticality to faults. The fanout of a value is defined as the number of dynamic instructions dependent on the static instruction that produces this value, i.e. uses the value as a source. Since a value with a high fanout will result in propagating the fault to more values, it is classified as being more critical to protect than one with a low fanout.

Figure 3 shows the effect of deploying oracular predictors at the data-only values with the top 100 fanout counts in which faults are hard-to-detect (faults in values that resulted in the *Detected-Long* and *SDC* categories in Figure 2(b)). For each data-only value, the fanout is computed by aggregating (set union) all dynamic instructions that use as source a data-only dynamic instance of the corresponding static instruction.[3] For each workload, Figure 3(a) shows the fraction of hard-to-detect faults identified by the 100 detectors. Since these detectors may also detect some faults that are masked by the application, we also measure their false positives rate as the fraction of detected faults that are masked. Figure 3(b) shows the false positive rate of the detectors for each workload.

Focusing on Figure 3(a), we see that 100 oracular detectors placed based on the fanout metric cover all the hard-to-detect faults in the data-only values for all applications but *ammp*. Since *ammp* has the many values that are data-only, 100 detectors do not suffice to detect all the hard-to-detect faults; this application requires approximately 500 detectors to detect all its hard-to-detect faults. The bars for *art* and *mesa* have 0% fanout coverage as *art* has no data-only values, and all faults injected into *mesa* are masked by the application. Thus, consistent with prior results, the fanout metric is effective in identifying critical data-only values.

Figure 3(b) shows however that the exact detectors placed need to be carefully designed in order to contain instances of false positives. As seen in the figure, the oracular detectors that we use, which identify *any* changes in data values at those locations, result in a high false positives rate of over 60% for several application, and over 38% for all but 2 applications. Previous work, however, demonstrated much lower false positive rates of under 4% [10] owing to the ability of their detectors to track dynamic data values.

These results demonstrate that although fanout is a useful metric to identify critical values which may cause hard-to-detect faults, selecting detectors purely based on fanout may result in high rates of false positives. Additional work that selects the most effective form of such detectors to track dynamic data values at the selected locations is required to lower these false positive rates further. There has been much work in this realm to identify faults by observing perturbations in data values [5], [11], [13], which may be leveraged for this purpose.

[3]Previous work aggregated fanout statistics across *all* instances of a static instruction with the assumption that multiple dynamic instances may share identical behavior in the presence of faults. We have found, however, that this need not be the case and hence aggregate our statistics only for those instances that produce data-only values.

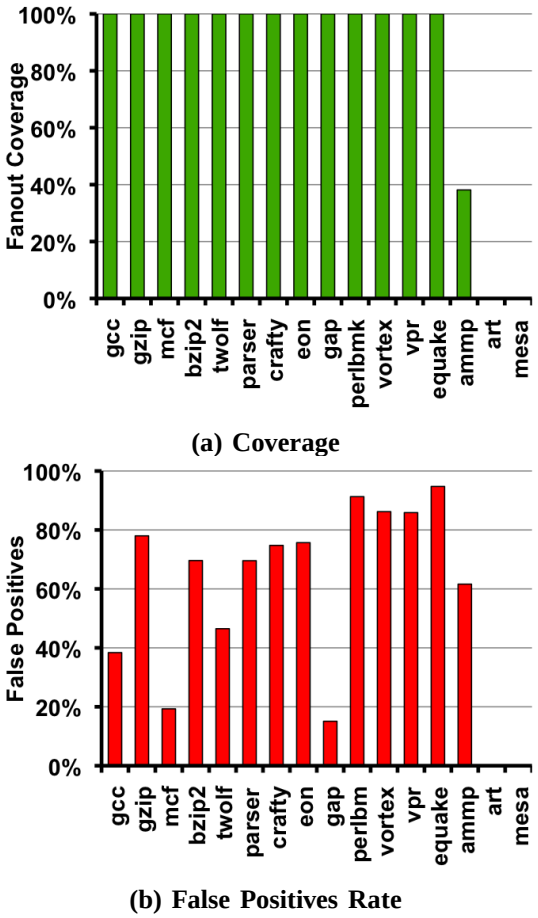


Fig. 3. Identifying critical data-only values with the fanout metric. While the fanout metric identifies most values that may result in hard-to-detect faults, it also results in a high rate of false positives.

IV. Conclusions and Future Work

This paper provides, for the first time, an intuition behind why symptom detectors, like SWAT, result in low SDC rates by studying application properties that lead to SDCs. It provides the key insight that faults in application values that corrupt control flow or memory addresses are easier to detect for symptom detectors than faults in other *data-only* values. It shows that only a small fraction of application values fall into this category, making symptom detection attractive in modern systems. It further demonstrates, through fault injection experiments, that the rate of SDCs with the SWAT detectors from transient faults injected into these data-only values is higher than the SDC rate from faults injected into random values. Additionally, in an attempt to predict the criticality of these data-only values, it explores previously proposed metrics to determine value criticality and finds that while there is a lot of promise in previously proposed metrics, the detectors may suffer from a high rate of false positives. Additional research is required to identify detectors that incur fewer false positives.

These constitute an important step towards understanding system resiliency from an application point-of-view. Research on future systems is leaning towards breaching traditional boundaries between hardware and software and incorporating application properties to improve the quality of the underlying hardware. An understanding of the relationship between application properties and resiliency will therefore be invaluable. This paper attempts to provide one such classification through its data-only values categorization and its associated analyses.

This work opens up several avenues for future work. First, the data-only values analysis presented in this paper only tracks the dependences of a given value for a fixed duration (of 10M instructions) due to infrastructure limitations. Tracking the dependences all the way to the application output would make the analysis more thorough and give deeper insight into application behavior. Second, the analysis of data-only values presented here need to be expanded to other classes of workloads such as media, client-server, etc. Finally, while the fanout metric showed much promise in identifying critical data-only values, the detectors need to be carefully designed to contain their false positives rate. Additional work to devise detectors that incur lower false positive rates is required.

References

[1] A. Benso et al. Data Critically Estimation In Software Applications. In *Proceedings of International Test Conference*, 2003.

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

[3] 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.

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

[5] M. D. Ernst et al. The Daikon System for Dynamic Detection of Likely Invariants. *Science of Computer Programming*, 2007.

[6] S. Hari et al. Low-cost Hardware Fault Detection and Diagnosis for Multicore Systems. In *Proceedings of International Symposium on Microarchitecture*, 2009.

[7] M. Li et al. Trace-Based Microarchitecture-Level Diagnosis of Permanent Hardware Faults. In *Proceedings of International Conference on Dependable Systems and Networks*, 2008.

[8] 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.

[9] A. Meixner et al. Argus: Low-Cost, Comprehensive Error Detection in Simple Cores. In *Proceedings of International Symposium on Microarchitecture*, 2007.

[10] K. Pattabiraman, Z. Kalbarczyk, and R. K. Iyer. Application-based metrics for strategic placement of detectors. In *Proceedings of Pacific Rim International Symposium on Dependable Computing*, 2005.

[11] P. Racunas et al. Perturbation-based Fault Screening. In *Proceedings of International Symposium on High Performance Computer Architecture*, 2007.

[12] L. Rashid, K. Pattabiraman, and S. Gopalakrishnan. Towards Understanding the Effects of Intermittent Hardware Faults on Programs. In *Proceedings of International Conference on Dependable Systems and Networks*, 2010.

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

[14] V. Sridharan and D. Kaeli. Eliminating Microarchitectural Dependency from Architectural Vulnerability. In *Proceedings of International Symposium on High Performance Computer Architecture*, 2009.

[15] 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.