Relyzer: Exploiting Application-Level Fault Equivalence to Analyze Application Resiliency to Transient Faults *

Siva Kumar Sastry Hari,[1] Sarita V. Adve,[1] Helia Naeimi,[2] and Pradeep Ramachandran[3]

[1]Department of Computer Science, University of Illinois at Urbana-Champaign, USA
[2]Intel Labs, Intel Corporation, Santa Clara, CA, USA
[3]Intel Corporation, India
{shari2,sadve}@illinois.edu, {helia.naeimi,pradeep.ramachandran}@intel.com

Abstract

Future microprocessors need low-cost solutions for reliable operation in the presence of failure-prone devices. A promising approach is to detect hardware faults by deploying low-cost monitors of software-level symptoms of such faults. Recently, researchers have shown these mechanisms work well, but there remains a non-negligible risk that several faults may escape the symptom detectors and result in silent data corruptions (SDCs).

Most prior evaluations of symptom-based detectors perform fault injection campaigns on application benchmarks, where each run simulates the impact of a fault injected at a hardware site at a certain point in the application's execution (application fault site). Since the total number of application fault sites is very large (trillions for standard benchmark suites), it is not feasible to study all possible faults. Previous work therefore typically studies a randomly selected sample of faults. Such studies do not provide any feedback on the portions of the application where faults were not injected. Some of those instructions may be vulnerable to SDCs, and identifying them could allow protecting them through other means if needed.

This paper presents Relyzer, an approach that systematically analyzes all application fault sites and carefully picks a small subset to perform selective fault injections for transient faults. Relyzer employs novel fault pruning techniques that prune faults that need detailed study by either predicting their outcomes or showing them equivalent to other faults. We find that Relyzer prunes about 99.78% of the total faults across twelve applications studied here, reducing the faults that require detailed simulation by 3 to 5 orders of magnitude for most of the applications. Fault injection simulations on the remaining faults can identify SDC causing faults in the entire application. Some of Relyzer's techniques rely on heuristics to determine fault equivalence. Our validation efforts show that Relyzer determines fault outcomes with 96% accuracy, averaged across all the applications studied here.

Categories and Subject Descriptors B.8.1 [*Hardware*]: Performance and Reliability—Reliability, Testing, and Fault-Tolerance

General Terms Design, Experimentation, Measurement, Reliability

Keywords Low-Cost Hardware Resiliency, Hardware Reliability Evaluation, Silent Data Corruption, Transient Faults, Architecture

1. Introduction

As process technology scales, the increasingly smaller devices become susceptible to a variety of in-field hardware failure sources; e.g., high-energy particle strikes (soft-errors), voltage droops, wear-out, and design bugs [4]. This increases the likelihood of a hardware failure in the field. Future systems must therefore handle such failures through in-field fault detection, diagnosis, repair, and recovery mechanisms to guarantee continuous error-free operation.

Hardware fault detection mechanisms form a crucial part in devising such reliability solutions. Traditional solutions use heavy amounts of redundancy (in space or time) to detect hardware faults. Owing to their prohibitive costs, such detection mechanisms are increasingly unacceptable for modern commodity systems. Instead, there is a growing recognition that a wide spectrum of the commodity space will accept only much lower cost solutions (in area, power, and performance), perhaps at the cost of tolerating very occassional failures.

Recently, there has been a surge of research in software-level symptom based fault detection techniques [5, 7, 13, 14, 16, 19, 20, 25] that provide promising such low-cost alternatives. These techniques detect only those hardware faults that corrupt software execution by monitoring for anomalous software behavior using simple, low-cost monitors. Despite the simplicity of their detectors, these techniques have demonstrated impressively high detection rates. Unfortunately, some fraction of faults do escape the detection mechanism and silently impact the correctness of the program output. Such faults are called silent data corruptions or SDCs. As an example, SWAT (SoftWare Anomaly Treatment) [8, 12, 22], a state-of-the-art reliability solution, reports an SDC rate of $<$0.5% across several (compute-intensive, media, and distributed client-server) workloads for both permanent and transient faults in all hardware units studied except the data-centric FPU.

By their nature, the effectiveness of symptom-based detection solutions depends on the application running on the system and when the fault manifests during the application's execution. Comprehensive evaluation of the effectiveness for a given application

* This work is supported in part by the Gigascale Systems Research Center (funded under FCRP, an SRC program) and the National Science Foundation under Grant CCF 0811693. Pradeep Ramachandran was supported by an Intel PhD fellowship and an IBM PhD scholarship. This work was done when Pradeep Ramachandran was a graduate student at the University of Illinois at Urbana-Champaign.

ASPLOS'12, March 3–7, 2012, London, England, UK.

requires studying the impact of all faults of interest injected at each cycle of the application's execution (one at a time). Below, when we refer to a *fault*, we include both the hardware site *where* a fault is injected as well as *when* it is injected in an application's execution (the application site). After injecting a fault (at a given cycle), the application must be allowed to run potentially to completion to determine if the fault is masked, results in SDC, or is detected (in the latter case, the execution may be stopped on detection). For application benchmarks with billions of instructions, this translates (conservatively) into trillions of fault injection runs for most benchmark suites and fault models of interest. Such a comprehensive fault injection campaign is clearly infeasible.

While most symptom-based detection techniques have been evaluated through fault injection campaigns [13, 14, 16], these studies bound the experiment time by studying a randomly selected sample of faults (typically of the order of thousands of faults per application) out of all the faults possible (more than a trillion faults for this study). While these methods may provide statistical guarantees SDC rates, certain faults that may be important to the application may be sampled out. Equally important, such statistical sampling provides virtually no feedback on which parts of the application remain vulnerable (other than the few instructions where faults were injected) and might need protection in other ways to reduce the SDC rate. With the ever-increasing constraints on power, performance, and area, such application-specific customization of resiliency solutions is desirable. Thus, for symptom-based detectors to become comercially adopted, it is important to provide firmer guarantees on SDC rates and to better identify the remaining vulnerable portions of the program to reduce the SDC rate.

This paper focuses on transient faults and seeks to dramatically reduce the number of faults to simulate while still preserving the ability to accurately determine the SDC rate and SDC vulnerable portions of the application. As mentioned above, the number of faults is large both due to the large number of hardware sites (*where*) and application execution sites (*when*) that a fault could be injected. There has been significant work in the literature to explore the impact of where in the hardware faults are injected, including understanding tradeoffs in injecting faults at the gate, microarchitectural, and architectural levels [10, 11, 17] and extensive work on notions of fault equivalence in the testing literature [1]. However, there is limited work on when in the application execution should faults be injected.

This paper presents *Relyzer*, a resiliency analyzer that systematically analyzes all (dynamic) application sites to determine a minimal set for when faults need to be injected.[1] Since our focus is not on the hardware sites, we choose two examples: transient single-bit flip faults in (architectural) integer registers and in output latches of address generation units.

Relyzer first lists all application sites that can be directly affected by our chosen transient faults. For some of these faults, Relyzer can directly predict their outcome (detection, masking, or SDC) through simple static analysis and dynamic profiling of the fault-free execution. These faults do not need detailed fault injection experiments. For the remaining faults, Relyzer primarily uses the insight that faults propagating "similarly" through the program are likely to result in similar outcomes. We propose novel heuristics based on static and dynamic control flow and data flow to capture the notion of "similar" for faults in different types of instructions. Using these heuristics, we categorize application fault sites into equivalence classes. We then select a representative from each equivalence class and thoroughly study it through a detailed fault injection experiment.

[1] We only consider single-threaded applications in this work and leave the exploration of multithreaded applications to future work.

Our results show that Relyzer significantly reduces the number of faults that require detailed fault injection experiments. Relyzer pruned about 99.78% of the total faults in the twelve applications studied here (three to five orders of magnitude reduction for all but one application). We validated the pruning techniques that use heuristics by matching the fault injection outcomes of the representative faults against outcomes of a sample of faults they each represent. Each pruning technique and fault model combination individually gave an accuracy of $>$92%, averaged across all studied applications. Overall, with the combination of all pruning techniques, Relyzer was able to correctly determine the outcomes of 96% of all faults, averaged across all twelve applications studied.

Overall, Relyzer significantly reduces the number of application fault sites that require thorough fault injection experiments, bringing them to a point where studying virtually all of them becomes practically viable. Further, it does so in a way that allows identifying the application sites vulnerable to SDCs (the equivalence classes whose representatives result in SDCs). To our knowledge, Relyzer is the first work to develop such a notion of fault equivalence for application fault sites (analogous to that of hardware fault equivalence). Section 5 describes the relationship with other work that has similar, but not identical, goals [2, 6, 18, 23].

2. Fault Pruning Techniques

Relyzer systematically analyzes all application fault sites and carefully selects a small subset for thorough fault injection experiments such that it can still estimate the outcomes of all the faults in the application. To achieve this goal, Relyzer applies a set of pruning techniques that are classified as *known-outcome* and *equivalence-based* pruning techniques. The known-outcome techniques largely use static (and some dynamic) program analyses to predict the outcome of a fault. The equivalence-based techniques prune faults by showing them equivalent to others using static and dynamic analyses and/or heurstics.

Relyzer first enumerates all the faults that can impact the application. We require a fault-free execution trace for a given application (and input) for this step. Each dynamic instruction instance in the trace forms a potential application fault site. At this site, we consider injecting faults in the hardware units that would be exercised by this instruction (one fault at a time). Since our focus is on the choice of application fault sites and not on exhaustively studying all hardware faults, we choose to study transient faults (single bit flips) in architectural integer registers and in output latches of address generation units. As an example, consider an add instruction with register operands g1, g2, and l1 as an instruction appearing in the dynamic instruction trace. We consider injecting single-bit-flips in the integer registers g1, g2, and l1 that are accessed by this instruction (one at a time, in different bits). Faults in the address generation unit will be considered only when it is exercised; i.e., in load and store instructions.

While enumerating the list of all application fault sites, Relyzer stores all the fault related information in a data structure called the fault database, shown in figure 1. In particular, it stores the identity of the static instruction (program counter), the hardware fault sites that can affect the instruction (e.g., names of registers), and the number of dynamic instances of the instruction. The rest of the fields in figure 1 apply to specific pruning techniques and are described with those techniques.

Relyzer next applies the fault pruning techniques on this initial set of faults. While the fault pruning is being performed, all the required information for successful computation of overall SDC rate and the SDC rate for each static instruction is logged in the fault database (figure 1). Additionally, some dynamic information to assist the later development of low-cost detectors can also be logged,

Fault Site			For known-outcome pruning techniques			For precise equivalence-based pruning techniques		For store- and control-equivalence pruning
Static Instruction (PC)	Faulty unit ID	Number of dynamic instances	Bit min	Bit max	Expected outcome	Equivalent instruction (PC)	Faulty unit ID	List of pilots

Pilot ID	Faulty unit ID and bit location	Population	Outcome	Extra information

Figure 1. Fault database. The first three fields in the first table store basic information regarding a fault site and the static instruction. The bit min and bit max fields indicate the amount of pruning performed by the known-outcome pruning techniques – faults in the bits between bit min and bit max are pruned and their predicted outcome is recorded in the expected outcome field. The equivalent instruction and the faulty unit ID fields store information for the precise equivalence-based pruning techniques. The information regarding pilots that are obtained by the store- and control-equivalence based pruning techniques is stored in a separate table. It also stores some extra information that can aid the development of low-cost detectors.

but we leave the exploration of such information and detectors to future work.

2.1 Known-outcome pruning technique

Bounding addresses: Transient hardware faults can make applications access memory locations that fall out of the range of the allocated address space. Such accesses are likely to result in detectable symptoms (e.g., fatal traps, segmentation faults, application aborts, and kernel panic). SWAT employs detectors specifically to detect such scenarios within recoverable latencies (e.g., out-of-bounds detectors [21]). We do not need injection experiments to identify the outcome of most such faults and can directly prune them as follows.

We determine the range of valid addresses, for both the stack and the 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 while profiling. This approximation only makes our technique conservative if we assume that the out-of-bounds detector can also detect faults in addresses that make accesses cross the global-heap boundary.

Once we identify the range of the valid addresses, we prune faults that would allow a memory instruction to access an invalid address (e.g., faults in high order bits of the address when the fault-free trace shows valid addresses are within lower order bits). This technique is applicable to memory instructions (both loads and stores).

Information for fault database: The known-outcome technique prunes faults by declaring them as detected. Hence, very little information is needed to be recorded in the fault database. We record the range of the bits that are pruned in the bit min and bit max field along with the estimated outcome (detected) in the expected outcome field of the fault database (figure 1).

2.2 Equivalence-based pruning techniques

This class of pruning techniques eliminates faults that are equivalent to each other from the initial set of faults and retains only the representative faults for thorough fault injection experiments. We further categorize pruning techniques in this section as *precise* and *heuristics-based*, based on whether they use accurate analyses or heuristics to form the equivalence classes.

2.2.1 Precise equivalence

Def-use analysis: A register definition is created whenever a register is used as a destination operand in an instruction. Faults in the definition of a register have similar behavior to that of faults in the first use of this definition. Therefore, we prune out faults in the definition and retain faults in the first use. Note that this technique prunes faults only in the definition and not in the uses. There can be multiple uses of a definition, and faults in different uses may have different fault propagation. Whenever a definition is pruned, we record the information of the first use at the definition. This allows relating the outcomes of the faults in the first use to those that of the definition's at a later stage.

Ideally, the destination register operands of all the instructions should be pruned by this technique. In our experiments, however, we prune faults in only those destination registers that have a first use within the same basic-block. Since we implemented this technique as a static program pass, accounting for this equalization in the fault database (i.e., associating faults in the first use to that of the definition's) was non-trivial for the cases where the def to first-use chains spanned across multiple basic blocks. Moreover, in the presence of conditional move operations, it was unclear whether a static pass can still prune faults without compromising on the precise association of a definition with the first-use. Hence, we limited ourselves to a conservative but precise implementation.

Information for fault database: The entries of the fault sites that are pruned by this technique record the information of the fault sites that represent them. In other words, the definition records the identity of the first use (the program counter of the instruction and the faulty unit id, as in figure 1).

2.2.2 Heuristics-based equivalence

Control-equivalence: This heuristic pruning technique uses the observation that faults propagating through similar code sequences are likely to behave similarly. It also uses the observation that a majority of the faults appear in code sequences that are executed many times. Consider a static instruction I with many dynamic instances in the fault-free execution under consideration. The pruning technique attempts to partition all these dynamic instances of I into equivalence classes, based on the control flow path followed after the dynamic instance.

It is convenient to describe and implement the algorithm at the basic block level. The technique uses the fault-free application execution to enumerate all possible control flow paths up to a depth n starting at the basic block that contains the instruction of interest. Depth is defined as the number of branch or jump instructions encountered. For the paths that were exercised multiple times in the execution, it randomly selects one dynamic occurrence, a pilot. It prunes all other unselected executions of such paths (population) and assumes that faults in those dynamic executions are represented by the selected ones (pilots). More precisely, a dynamic instruction instance on a pilot path serves as a pilot for other instances with the same PC on the other paths in its population.

Figure 2 explains through an example how this pruning technique selects pilots. The figure presents a control flow graph of a small program, with the basic blocks represented by the black and grey circles with numbers on their sides. Assume the grey basic block is not exercised by the dynamic execution of interest. Assume $n = 5$ (depth until which control flow is tracked). Suppose we are interested in finding the representative pilots for an instruction in basic block 1. We enumerate all control flow paths starting at basic block 1 up to a depth of 5 that are executed in the dynamic

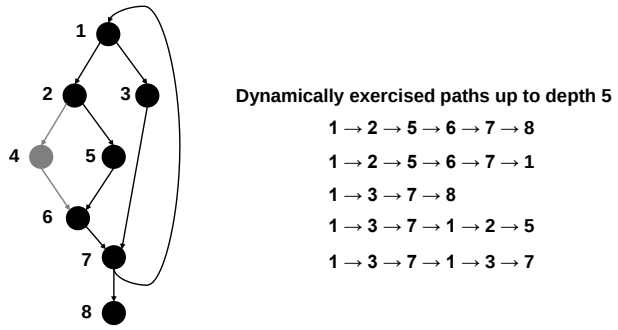


Figure 2. Control-equivalence. The figure shows a CFG for a small program starting at basic block 1 and ending at basic block 8. We enumerate all dynamically exercised control paths up to a depth, say 5. Here basic block 4 (showed in grey) never gets exercised. Therefore control flow paths through this node do not appear on the list of dynamically exercised paths. The executions along each of these paths form the equivalence classes for similar fault outcomes.

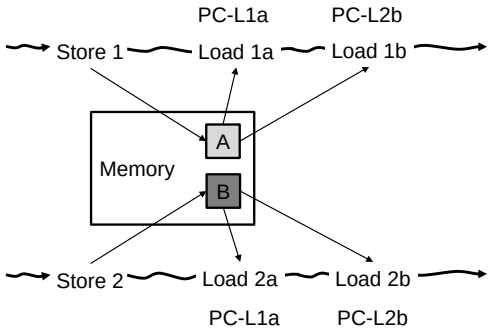


Figure 3. Store-equivalence. Store 1 and Store 2 are two store instructions from the same static instruction writing to addresses A and B respectively. Load 1a with program counter PC-L1a and Load 1b with program counter PC-L1b are two load instructions reading the value from address A. Similarly, Load 2a and Load 2b are two loads from address B with program counters PC-L2a and PC-L2b respectively. The store-equivalence heuristic requires that PC-L1a equal PC-L2a and PC-L1b equal PC-L2b.

fault-free execution of interest. Basic block 4 is never executed and hence it does not appear in the list of dynamically exercised paths. We identify each path as forming a new equivalence class. There will be potentially many instances of such paths in the dynamic execution trace. We randomly select one dynamic execution sequence for each equivalence class and name it as the pilot for that class. As mentioned before, a dynamic instruction instance on a pilot path serves as a pilot for other instances with the same PC on the other paths in its population.

We apply this technique to prune faults in all instructions other than stores and those that affect stores within a basic block. This is because the propagation of a fault in a store also depends on the addresses of the loads in the control flow path taken (only loads to the same address as the store will propagate the fault). The next technique described deals with this distinction. Exceptions to the above are a few SPARC specific instructions; namely, *save*, *restore*, *call*, *return*, and *read state register*. In this study, we do not inject faults in these instructions and therefore do not consider them any further. We also do not inject faults in dead instructions and do not consider those any further either.

Overall, control-equivalence has the potential of pruning a large fraction of the faults by softening the constraint on evaluating all dynamic occurrences from a specific code section.

Store-equivalence: A fault in a store instruction propagates through the loads that read the faulty values. Load addresses are not entirely captured by the control flow path taken after the store. We therefore developed an alternate heuristic, called store-equivalence, for faults in store instructions or in instructions that a store depends on within the same basic block. This heuristic captures the fault propagation behavior by observing the addresses that a store writes in a fault-free execution and recording all read accesses to this address. It treats the faults in stores differently whenever a different permutation of load instructions read the stored value.

Figure 3 illustrates our heuristic with an example. Consider Store 1 and Store 2 as two dynamic store instruction instances from the same static instruction. To determine if the faults in these two store instructions will have the same outcomes, we examine all the loads that return the values written by these stores in the fault-free execution, i.e., Load 1a and Load 1b for Store 1 and Load 2a and Load 2b for Store 2 from the figure. We first check whether the number of such loads is the same (two for each store in the figure). If this is the case, then we check whether the static instructions (program counters) of the corresponding loads are the same; e.g., if the program counters of Load L1a and Load L2a are the same and if those of Load L1b and Load L2b are the same in the figure. If these match, then we conclude that the two dynamic store instructions are very likely to have similar fault outcomes and we place them both in the same equivalence class.

Information for fault database: We record the information regarding the pilots in the fault database (figure 1). In particular, we record the unique pilot id and the size of the population it represents. We can also record additional information such as the control path (sequence of instructions) this pilot represents for control-equivalence or the usage pattern of the stored value (e.g., number of loads and their program counters) for store-equivalence in the extra information field. This information may be helpful for the applications of Relyzer (e.g., understanding and protecting SDC causing fault sites).

2.3 Other pruning techniques

We considered several other pruning techniques, but they provided limited benefit. We include them here for completeness.

Bounding branch targets (known-outcome category): Analogous to the bounding addresses case (known-outcome pruning technique), a fault that causes a control instruction to jump to a location that is not in the application instruction space is likely to result in a detectable symptom (e.g., SWAT's fatal trap and app-abort detectors or an out-of-bounds detector analogous to that for data addresses can detect such faults).

The address range that contains all possible targets can be obtained by noting the start and the end of the text section of an application. Typically, the text section is small (for applications with under million instructions; i.e., under 32 bits) and hence a large fraction (over 50% on 64-bit machines) of faults in branch targets can be predicted as detected and pruned by this technique. This bounding technique may not be directly applicable to jumps to shared libraries because the registers used by these operations may already contain addresses that are out of the text section.

We performed an optimistic experiment on the studied fault models and applications and found that it only provides a pruning of 0.5%. Hence, we do not include the bounding branch targets technique in our study here. However, this technique may be ef-

fective for other faults models; e.g., faults in immediate operands. Many branch instructions specify PC-relative displacements as immediate operands, and could benefit from this technique.

Constant-based bit masking (known-outcome category): In some logical operations, only a subset of the bit locations in the source operands are used to produce the destination register value. Hence, faults in the source operand's bit locations that are usually discarded will be masked. Currently we apply this technique only on logical shift operations, where the shift count is a constant. We prune faults in the source register's bit locations that are not used to produce the destination register value and treat them as masked.

This technique provides a modest benefit for the unoptimized version of our applications. We, however, do not report it for the optimized version of our applications because preliminary experiments showed that it provides insignificant benefits.

Several other instruction-specific pruning techniques (similar to the one above) have a potential of providing added pruning. This is a part of our future work.

Constant-based equivalence (precise-equivalence category): We applied the principal of constant propagation to prune faults from operands of instructions that use constants. For such instructions, the effect of a fault in the source register (non-constant) operand can often be studied directly in the destination operand; therefore, we can prune faults from such source operands. In our implementation, this pruning technique is currently limited to only those logical operations where a single-bit fault in the source operand propagates as a single-bit fault in the destination (e.g., logical `xor`). This technique also provides negligible benefit for the optimized applications. Hence we report it only for the unoptimized applications.

Information for fault database: The first two techniques above qualify as known-outcome based because they prune faults by declaring them as detected or masked. Hence, similar to the bounding addresses case, we record the range of the bits that are pruned in the bit min and bit max field along with the estimated outcome in the expected outcome field of the fault database (figure 1).

The third technique above is similar to def-use analysis. Consequently, the fault database is updated with the information about the fault equalization, i.e., the information of the destination operand (program counter of the instruction and the faulty unit id in figure 1) is recorded at the source operand.

2.4 Implementation

Relyzer implements the pruning techniques using static and dynamic program analyses. For example, the def-use and known-outcome pruning techniques are implemented largely as a static program pass (they use some basic information from the dynamic fault-free execution profile). Store- and control-equivalence techniques, on the other hand, largely use the dynamic information.

As a first step, Relyzer initializes the fault database and computes the initial set of faults. The information about the number of dynamic instances of each static instruction is required to compute the size of the initial set of faults and is obtained through a dynamic profile of the application. Relyzer then applies the first pruning technique – known-outcome address bounding, which is implemented as a static program pass. This technique requires the knowledge about the boundaries of stack and heap addresses accessed during the entire course of the program and this information is obtained through dynamic profiling. Relyzer next performs the def-use analysis (the precise equivalence based pruning technique), which is also implemented as a static program pass. For applications with unoptimized codes, it also applies the two constant-based pruning techniques at this step.

Once these static techniques are applied, Relyzer prepares the application codes for dynamic analysis for the heuristics-based equivalence pruning techniques (control- and store-equivalence). It first labels all static instructions to indicate which (if any) of the heuristics-based pruning techniques will be applicable to them. For instructions within a given basic block, all stores and the instructions that any store is dependent on are labeled to be pruned by store-equivalence. Since the dynamic store-equivalence based analysis is performed only on store instructions, the identity of the store instruction is recorded with the instructions that affect this store (in other words, the instructions that any store depends on also record the identity of the store instruction). Once the instructions for store-equivalence labeled, all other remaining instructions (with the earlier mentioned exceptions) are marked to be pruned by control-equivalence.

Relyzer then profiles the instructions and memory in more detail to obtain dynamic control- and store-equivalence classes as discussed in Section 2.2. It next uses the store-equivalence classes (for store instructions) and associates them with the instructions that recorded the identity of these stores (as mentioned above) to obtains the respective classes for all instructions that stores depend on. As a last step, Relyzer computes the remaining number of pruned faults by analyzing the updated fault database.

2.5 Computing SDC rates

Once all the pruning techniques are applied, the fault injection experiments on the remaining fault locations are performed. The SDC rate of the application can now be computed by using the information stored in the fault database (figure 1) as follows.

We start by computing the SDC rate for a static instruction, examining each faulty unit affecting the instruction. If a heuristics-based equivalence technique is employed for the unit, then we examine the outcomes of the instruction's pilots for each bit of the faulty unit not already pruned by the known-outcome techniques.[2] For each pilot and faulty bit combination that produces an SDC, we use the pilot's total population to increment a count of the total SDCs attributed to the faulty unit for this instruction.

For a faulty unit on which a precise-equivalence pruning technique is applied, we use the SDC count computed for the static instruction and faulty unit combination that was determined to be equivalent to the fault being examined (e.g., first use of a definition).

The total SDC count for a static instruction is the sum of the SDC counts for all its faulty units as obtained above. The SDC rate for a static instruction is its SDC count divided by the total number of initial faults attributed to this instruction. The SDC rate of an application is now simply the total SDC count of all the static instructions divided by the total number of initial faults in the application.

3. Methodology

3.1 Workloads

We implemented the pruning techniques described in Section 2 for single-threaded applications compiled for the SPARC V9 [26] architecture, assuming the hardware fault models previously described. We selected twelve applications – four each (randomly selected) from the SPLASH-2 [27], PARSEC [3], and SPEC CPU2006 [9] benchmark suites. Table 1 provides a brief description of these applications, including the inputs used, the dynamic instruction count, and the number of faults prior to applying any

[2] We assume the known-outcome techniques do not determine SDCs. The SDC count from faults pruned by future such techniques can be easily incorporated.

pruning. We do not include the initialization and the output phases of the applications in our study – these phases are usually dominated by file reads and writes, memory allocation and deallocation, etc. We found that the effectiveness of the developed pruning techniques varies significantly depending on whether the applications are optimized. We focus our results primarily on the optimized versions of the applications. Section 4.1, however, briefly summarizes the impact of optimizations for a subset of the above applications. The dynamic instruction counts and the number of faults in table 1 pertain to the optimized version of the applications.

3.2 Fault injection framework

As previously mentioned, the fault models we study are single bit flips in architectural integer registers and in the output latches of the address generation units (for loads and stores). Our fault injection simulation infrastructure uses a full system simulation environment comprising of Wind River Simics [24] and the GEMS microarchitectural and memory timing simulator [15], running our applications on the OpenSolaris operating system and compiled to the SPARC V9 ISA. This framework is similar to that used in the previous work on SWAT (e.g., [13]) with some modifications.

Our framework allows us to inject faults at any point in the application execution. This is the chosen application fault site, as represented by a dynamic instruction in the fault-free execution. To inject a fault, we start the application and execute it in functional mode (Simics-only) up to 500 cycles before the chosen application fault site. Then we start detailed timing simulation (Simics+GEMS) and inject the fault when the application fault site is reached. Thus, for address generation unit faults, we flip the specified bit in the unit's output latch when it generates the address for the specified dynamic instruction. For integer register faults, we flip the specified bit in the specified register when the specified dynamic instruction reads the register (for a source) or writes the register (for a destination). The flipped bit retains its state until the latch or register is overwritten. We then simulate the application for another 500 instructions in the detailed mode before switching to the functional mode and running it to completion.

We check for all SWAT symptoms [21] (fatal traps, application aborts, and kernel panics) in the detailed mode and a reduced set of symptoms (fatal traps, kernel panics, and system error messages) in the functional simulation phase. If a symptom is detected or a timeout condition is met (the application executes more than twice its expected runtime before producing the output), then we terminate the simulation and the outcome is recorded as detected. Otherwise, the output of the application is collected and compared with the fault-free output. We record the outcome as masked or an SDC depending on whether the two outputs are or are not the same respectively.

Note that when we inject a fault, there is always an instruction that consumes a faulty value or uses a faulty address. Thus, compared to pure microarchitecture-level injection simulations, we see no microarchitectural masking and very limited architectural masking. This is by design since we wish to maximize the injections that might lead to SDCs.

3.3 Pruning techniques

The pruning techniques require both static and dynamic analyses of the application. The static analyses study the binary and extract several properties that are either directly applied towards fault pruning or are later used by the dynamic technique. Since our fault injection infrastructure is developed for the SPARC V9 ISA model, we restrict our study to SPARC V9 binaries. We could not find any publicly available tools to analyze SPARC binaries, so we developed our own static binary analyzer that performs basic control flow and data flow analyses.[3] Using this static infrastructure, we traverse the application and create the set of all transient fault sites. We then apply the static pruning techniques, compute the pruned fault set, and collect information for dynamic analyses. The dynamic analyzer profiles the branches, the memory access patterns (for store-equivalence technique), instruction control flow patterns (for control-equivalence technique), etc. We use Wind River Simics [24] to implement these dynamic profilers. Finally, we use the information from both the static and dynamic analyses to generate the final pruned fault set.

For store-equivalence pruning, we dynamically observe every store instruction, the addresses they write to, and record all loads that read the stored value (as explained in Section 2.2). For mcf, however, we record only the first ten loads instead of all loads for forming the store-equivalence classes such that our store-equivalence algorithm finishes in a reasonable time of <10 hours.

To quantify the impact of the pruning techniques, we report the percentage of total faults that are pruned (in total and by the individual techniques) and the absolute number of remaining faults (pilots) that must be simulated to determine the resiliency of an application.

3.4 Validating pruning techniques

The control- and store-equivalence based fault pruning techniques use heuristics and require validation. Each of these pruning techniques chooses a dynamic instruction (*pilot*) to represent the outcome of several other dynamic instructions (the *population*). We quantify the validity of these techniques by quantifying the extent to which the pilots correctly represent the population. For example, suppose the injection of a fault in a pilot results in masking the fault. Suppose the injection of an analogous (hardware) fault in all members of the population results in 98% of the outcomes being masked and 2% detected or SDC. Then we say that the prediction rate of the pilot is 98%. The overall prediction rate for the pruning techniques is the weighted average of the prediction rate for all the pilots for that technique, weighted by the fault populations represented by the pilots.

To find the exact misprediction rate, ideally, we would run fault injections for all the pilots and all their associated populations. Simulating this combination is clearly prohibitive in simulation time. To reduce this time, we first restrict our validations only to the optimized applications. Further, for a given pilot, we randomly sample its population to determine the prediction rate. We select the sample size such that the 99% confidence interval for prediction is within 5% of the actual prediction rate.[4] We then inject transient faults in the pilot and the selected samples to obtain the prediction rate. Ideally, we would inject faults in all bit locations in the appropriate faulty units for the pilot, but the simulation time would be prohibitive. We instead injected faults in every 8th bit (bits 0, 8, 16, 24, 32, 40, 48, and 56 for a 64-bit register or the output latch of the address generation unit) that was not already pruned by the known-outcome pruning technique (e.g., if the known-outcome pruning technique prunes higher-order 32 bits, then we inject faults only in bits 0, 8, 16, and 24).

Sampling the population for a given pilot still leaves the problem that there are many pilots, each of which would require a

[3] We use the dynamic branch profile to create a correctly connected control flow graph because jump and link instructions create broken edges in the graph that may not be completed through static information alone.

[4] The pilot requires only one fault injection experiment to obtain the outcome A. We can formulate the fault injection experiments for the population as a Bernoulli trial with outcomes being either A or not A. Assuming all the experiments are independent, we can apply the principals of confidence intervals used for normal distributions.

Benchmark Suite	Application	Description	Input	Num. Dynamic Instructions	Num. Faults
Parsec 2.1	Blackscholes	Calculates prices of options with Black-Scholes partial differential equation	sim-large	22.3 Million	1.9 Billion
	Fluidanimate	Simulates an incompressible fluid for interactive animation purposes	sim-small	611.4 Million	102.5 Billion
	Streamcluster	Solves the online clustering problem	sim-small	1.44 Billion	106 Billion
	Swaptions	Computes prices of a portfolio of swaptions using Monte Carlo simulations	sim-small	922.2 Million	97.3 Billion
Splash-2	FFT	1D Fast Fourier Transform	64K points	548 Million	48.7 Billion
	LU	Factors a matrix into the product of a lower & upper triangular matrix	512×512 matrix 16×16 blocks	402.8 Million	33.2 Billion
	Ocean	Simulates large-scale ocean movements based on eddy and boundary currents	258×258 ocean	358 Million	21.7 Billion
	Water	Evaluates forces and potentials that occur over time in a system of water molecules	512 molecules	504.3 Million	36.6 Billion
SPEC-Int 2006	Gcc	Based on gcc Version 3.2, generates code for Opteron	test	3.8 Billion	500.4 Billion
	Libquantum	Simulates a quantum computer running Shor's polynomial time factorization algorithm	test	235.4 Million	27.4 Billion
	Mcf	Vehicle scheduling using a network simplex algorithm	test	4.57 Billion	485.4 Billion
	Omnet++	Uses the OMNet++ discrete event simulator to model a large ethernet campus network	test	1.35 Billion	146 Billion

Table 1. Applications studied. The number of dynamic instructions and faults pertains to the optimized versions of the applications.

large number of simulations for validation. We therefore restricted the number of pilots such that it was feasible to simulate all of them (and their sampled populations) in the available time. We selected enough pilots such that the total number of fault injections we had to perform for validation (for pilots and the population) was over one million for each of control- and store-equivalence (1,378,000 for control and 1,093,000 for store) across all fault models. In particular, for validating control-equivalence, we performed approximately 1,092,000 and 286,000 injections for integer register and address generation unit fault models respectively. For store-equivalence, he corresponding number of injections are 835,000 and 258,000. Further, each selected pilot represented a population of at least 1,000. For the 99% confidence interval, our average validation results for control-equivalence pruning have error bars of 1.84% and 3.67% for the integer register and address generation unit fault models respectively. For store-equivalence pruning, the corresponding error bars are 2.85% and 4.61%.

4. Results

4.1 Effectiveness of pruning techniques

4.1.1 Overall pruning effectiveness

Tables 2(a) and (b) show the overall effectiveness of Relyzer's pruning techniques by presenting the percentage of total faults pruned for the optimized and unoptimized applications respectively. The tables also show the absolute number of total faults and the faults remaining after pruning. The applications are ordered according to the total number of original faults.

For optimized applications, we find that Relyzer prunes an aggregate of 99.78% of all the studied faults across all applications. The total number of faults that need to be simulated reduces from 1.6 trillion to 3.52 billion, a three to five orders of magnitude reduction for all applications except mcf. The lowest pruning rate for a single application was 99.43% (for mcf) while most applications saw a pruning rate of 99.99%. For mcf,[5] two stores observed a pruning of 20%, bringing down mcf's overall pruning rate. The number of remaining faults in these two stores and the instructions that these stores depend on alone accounted for 83% of the total remaining faults for mcf.

4.1.2 Pruning effectiveness of individual techniques

Figures 4(a) and (b) show the effectiveness of Relyzer's individual pruning techniques for the optimized and unoptimized applications respectively. The stacks in each bar show the contributions of the individual pruning techniques when applied in the order shown (bottom to top) for all the faults in the application. There is no stack for constant-based pruning techniques in part (a) because our preliminary experiments showed these techniques provide limited benefit for those applications.

Focusing on the optimized applications, we found that the known-outcome pruning technique pruned an average of approximately 27% of all the faults. Def-use analysis prunes 15% of all the faults on average. Thus, the above mostly static techniques alone provided approximately 42% of pruning across our applications. Control-equivalence is overall the most effective individual technique for these applications, providing 48% of the pruning on average. Finally, store-equivalence technique pruned about 10% of all the faults.

The unoptimized applications show slightly different behavior. First, store-equivalence provides notably more pruning than in the optimized applications. A likely reason is that there are more memory operations in unoptimized codes since they use the stack heavily and the registers poorly. Moreover, these store operations are often represented by a small number of pilots because they observe few permutations of loads during store-equivalence pruning. Second, the constant-based techniques provide significantly

[5] For forming the store-equivalence classes for mcf, we accounted for the first ten loads instead of all loads so that our algorithm finished in a reasonable time of $<$10 hours.

Application	Initial faults (in billions)	Total pruning	Remaining faults (in millions)
Blackscholes	1.9	99.99%	0.07
Ocean	21.7	99.99%	2.9
Libquantum	27.4	99.98%	4.1
LU	33.2	99.99%	1.1
Water	36.6	99.99%	2.1
FFT	48.7	99.99%	0.3
Swaptions	97.3	99.99%	0.6
Fluidanimate	102.5	99.91%	92
Streamcluster	106	99.99%	8.6
Omnet++	146	99.99%	2.2
Mcf	485.4	99.43%	2,781
Gcc	500.4	99.88%	627.5

(a) Optimized applications.

Application	Initial faults (in billions)	Total pruning	Remaining faults (in millions)
Blackscholes	4.01	99.99%	0.03
FFT	61.18	99.99%	0.16
Libquantum	127.03	99.93%	3.40
LU	175.36	99.99%	0.80
Swaptions	318.66	99.99%	0.08

(b) Unoptimized applications.

Table 2. Effectiveness of Relyzer's pruning on (a) optimized and (b) unoptimized applications. The applications are in increasing order of total number of original faults.

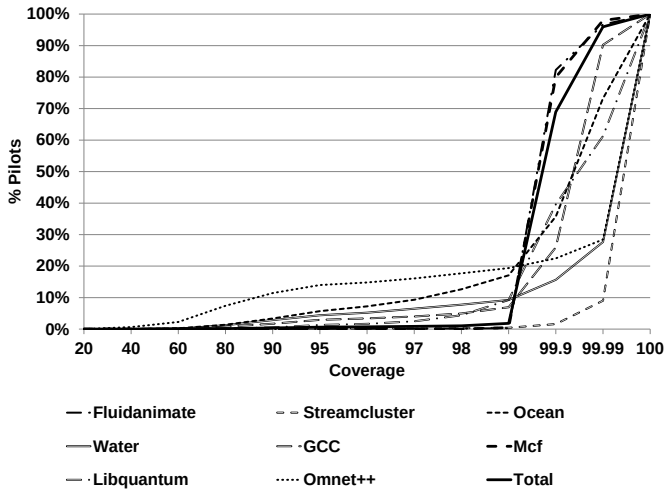


Figure 5. The percentage of pilots (y-axis) required to provide a desired amount of fault coverage (x-axis) for optimized applications. The x-axis shows the percentage of the total initial number of faults that are covered by the corresponding percentage of pilots. This includes the faults from the known-outcome category which are always considered covered. Note that the scale on the x-axis is not linear. Only individual applications that have more than 1 million remaining (not pruned) faults are shown, along with a curve for the aggregate faults across all applications.

more pruning for the unoptimized applications (about 6.5% of total faults). We observed that the number of remaining faults increases significantly (by about 100%) when this technique was excluded for the unoptimized applications. However, it had negligible impact on optimized applications. Overall, figure 4 shows significant differences in the relative effectiveness of the pruning techniques between the optimized and unoptimized codes, showing that compiler optimizations do impact the behavior of fault propagation.

4.1.3 Trading off simulation time with coverage

Although Relyzer is able to prune faults effectively, there are still a relatively large number of remaining faults that need to be simulated, especially for the longer applications. Relyzer allows a systematic method to trade off simulation time with coverage, revealing sweet spots that dramatically reduce simulation time with modest reduction in coverage.

Figure 5 shows the percentage of pilots (y-axis) needed to provide a desired coverage of the faults across the entire application (x-axis) after applying all pruning techniques for the optimized applications. These pruning techniques include the known-outcome class, which is considered to be always covered. For readability, we plot only the individual applications that had more than 1 million remaining faults that need simulation. For the full picture, we also plot the data for the total number of faults.

It is evident from the figure that only a small fraction of the pilots cover most of the faults. For example, 99% of all the faults across all the studied applications can be covered by 1.81% of the pilots. This corresponds to approximately 32 million faults. We can reduce this set further by compromising on the bit locations; e.g., injecting a fault in only every eighth bit of a given dynamic instruction, as in our validation experiments. With our existing simulation speeds, this set of faults can be simulated in approximately 11 days on a cluster of 200 cores.

We observed similar results for unoptimized codes as well, but do not present them here because most of the unoptimized applications we studied have under 1 million remaining faults.

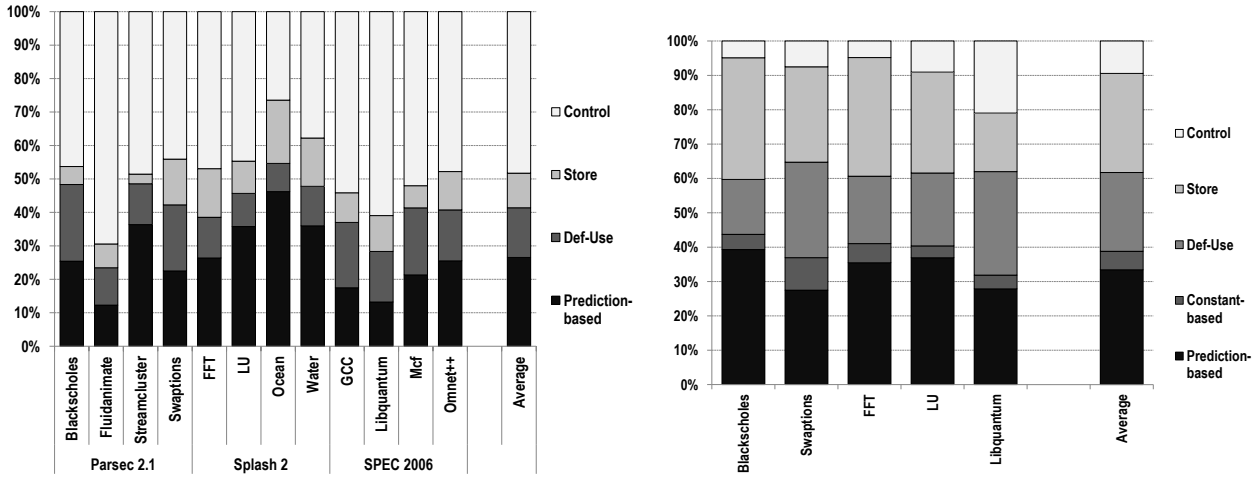
4.2 Validation of heuristics-based pruning techniques

4.2.1 Prediction rate for control- and store-equivalence

We validated the heuristics-based pruning techniques, namely control- and store-equivalence, for the optimized applications as described in Section 3.4. Figure 6 shows the prediction rate of the pilots for all twelve applications and both the studied hardware fault models (integer register or *reg* and output latch of address generation unit or *agen*). The combined bar for each application shows the observed prediction rate across all studied fault models after applying all pruning techniques. For each application, the combined bar is the average of the prediction rate of each pruning technique and fault model combination, weighted by the fraction of faults pruned by that combination. Specifically, in addition to accounting for control- and store-equivalence, this bar also accounts for faults pruned by def-use pruning (by associating a def-use pruned fault's prediction rate with that of its representative faults' rate). It also accounts for known-outcome based pruning, assuming a 100% prediction rate for that technique.

Figure 6 shows that the pilots selected through control-equivalence predict the outcome of their populations with an average (across all applications) accuracy of 95.7% for reg faults and 94.5% for agen faults. The pilots selected by store-equivalence predict their population's outcomes with an average accuracy of 95.4% for reg faults and 92.6% for agen faults. The figure also shows that for each individual application, Relyzer predicts the outcome across all fault models and pruning techniques with an accuracy of $>$91% (shown by the *combined* bar). This prediction rate averaged across all applications is 96%.

Integer register faults observe a prediction accuracy of approximately 90% or higher for all applications except Ocean. On the



(a) Optimized applications

(b) Unoptimized applications

Figure 4. Effectiveness of the individual pruning techniques for (a) optimized and (b) unoptimized applications.

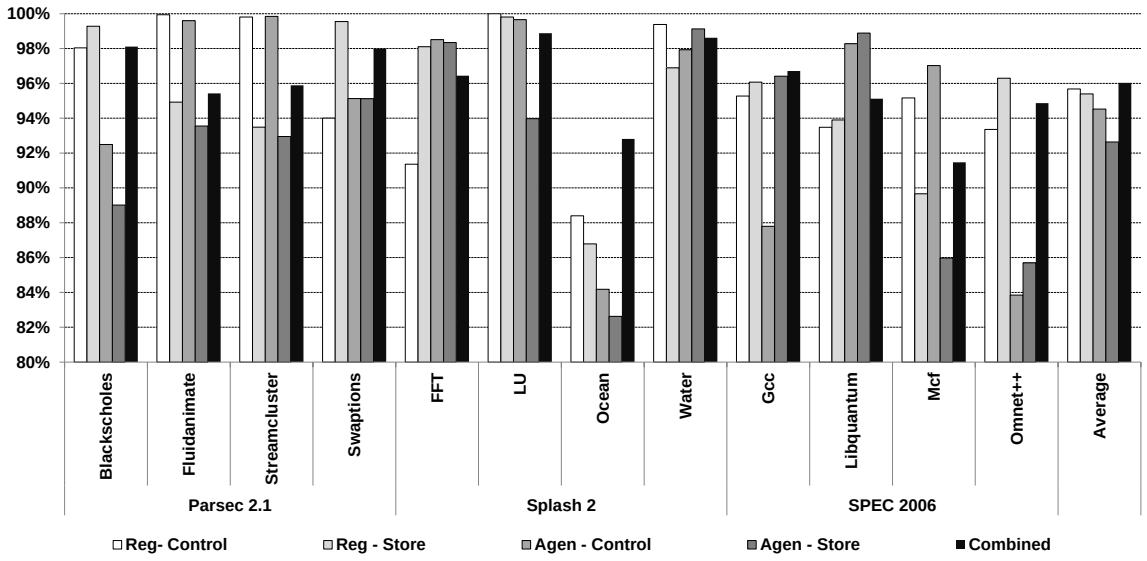


Figure 6. Validation of control- and store- equivalence for integer register (reg) and output latch of address generation unit (agen) faults for optimized applications. The *combined* bars for each application show the prediction rate across all fault models and pruning techniques.

other hand, agen faults showed <90% (the lowest is about 82%) for some cases for five applications – Blackscholes, Ocean, Gcc, Mcf, and Omnet++. We examined a few of these cases to understand why the prediction rate was not higher, and believe many of these can be eliminated by refining our heuristics.

For example, for Omnet++, a notable contributor to the mispredictions with control-equivalence for agen faults was a load instruction that should have been labeled for store-equivalence pruning. The instruction directly affected a store (and nothing else), but in the next basic block. Since our analysis looks only within the basic block for such data dependences to select instructions for store-equivalence pruning, it could not find this dependency. We plan to extend our static techniques in the future to enable correct labels in such cases.

As another example, we examined Blackscholes for store-equivalence pruning for agen faults. The major contributor to the misprediction rate was a load instruction loading values from faulty addresses. In several cases, it so happened that the faulty and correct address had the same value; therefore, the fault was masked rightaway. On the other hand, sometimes this was not the case, and the fault led to an SDC. A common pilot represented both classes of cases, leading to mispredictions. The fundamental issue is that our heuristics do not examine the faulty value. For Blackscholes, we could easily modify our implementation so that during our profiling step (fault-free execution), for every potential fault in a load address, we check the faulty address to determine if the value is different. This leads to a known-outcome based technique that can immediately determine if such a fault would be masked.

Examining the mispredicted cases in Ocean for agen faults pruned by store-equivalence exposed a more difficult limitation of Relyzer. A store instruction with a faulty address was one of the major sources of the high misprediction rate in Ocean. Such a store

corrupts the intended address by not writing the value from the source register and also the faulty address by writing an unintended value. We found the root cause of the misprediction to be the writing of the source register value in the faulty address. Since Relyzer cannot examine fault propagation through faulty addresses, this becomes a fundamental limitation and overcoming it is an interesting future direction.

4.2.2 SDC vs. non-SDC prediction rate

A key application of Relyzer is identifying SDC causing fault sites; therefore, we would like to ensure that Relyzer's prediction rate for SDC causing faults is also high. The data in figure 6 showed the prediction accuracy for masked and detected faults as well. Here, we distinguish only between SDC and non-SDC outcomes, treating the masked and detected outcomes the same. With just the SDC and non-SDC categories in mind, we revisited the validation results in figure 6. We observed that the average (across all applications) prediction rate for control-equivalence for reg and agen faults is 96.6% and 96.2% respectively, slightly higher than the overall prediction rate which also distinguished between the two non-SDC outcomes. Similarly, the average prediction rate for store-equivalence for reg and agen faults is also a higher 95.9% and 94.9% respectively.

4.2.3 Outcomes for fault injections in pilots

We next show that our choice of pilots is not biased towards any specific fault outcome. We plotted the outcomes from the fault injection experiments of the pilots that were selected for validation (described in Section 3.4). Figure 7 shows the distribution of the outcomes for reg faults in part (a) and agen faults in part (b). These two figures represent just 3,298 fault injection experiments. Each bar in figure 7 has between 129 and 320 injections for part (a) and between 27 and 102 for part (b). Even with such small sample sizes, the fault outcomes have a significant fraction of SDCs. In aggregate, about 23% of the pilots result in SDCs for both reg and agen faults. This result indicates that Relyzer can be effective in finding SDC causing fault sites.

4.2.4 Need for multiple pilots per static instruction

Our control- and store-equivalence heuristics partition the dynamic instances of a static instruction into multiple equivalence classes. Here we provide evidence for the need for such partitioning. In other words, we show that a naive algorithm that chooses just one pilot for each static instruction may be insufficient. Figure 8 shows the fraction of static instructions that are represented by different numbers of pilots. It shows that across all applications, at least 47% of the static instructions that are marked to be pruned by either store- or control-equivalence are represented by at least two pilots.

To provide further evidence for the need for such categorization, we examined one of the static instructions in LU that had multiple pilots representing it (12 in this case). We found that 8 of them resulted in SDCs (together representing 70% of the dynamic instances of that instruction) and 4 in Masking (all these pilots showed very high prediction rates).

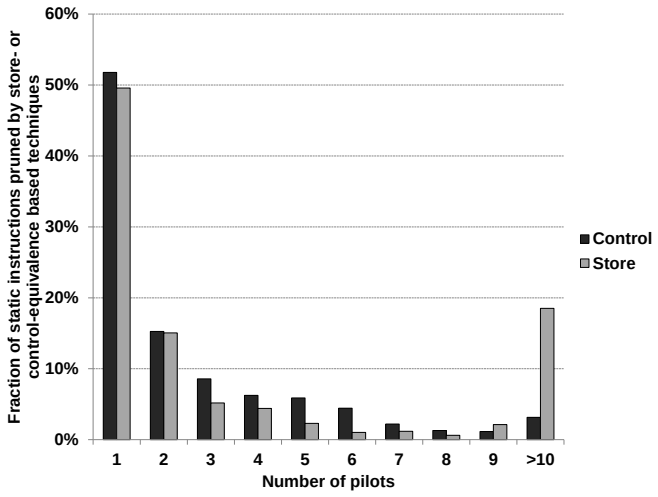


Figure 8. Pilots per static instruction. The figure shows the fraction of static instructions (averaged across all the optimized applications, pruned either by store- or control-equivalence) that are represented by a specified number of pilots. For example, 15% of the static instructions that are pruned by store-equivalence are represented by 2 pilots.

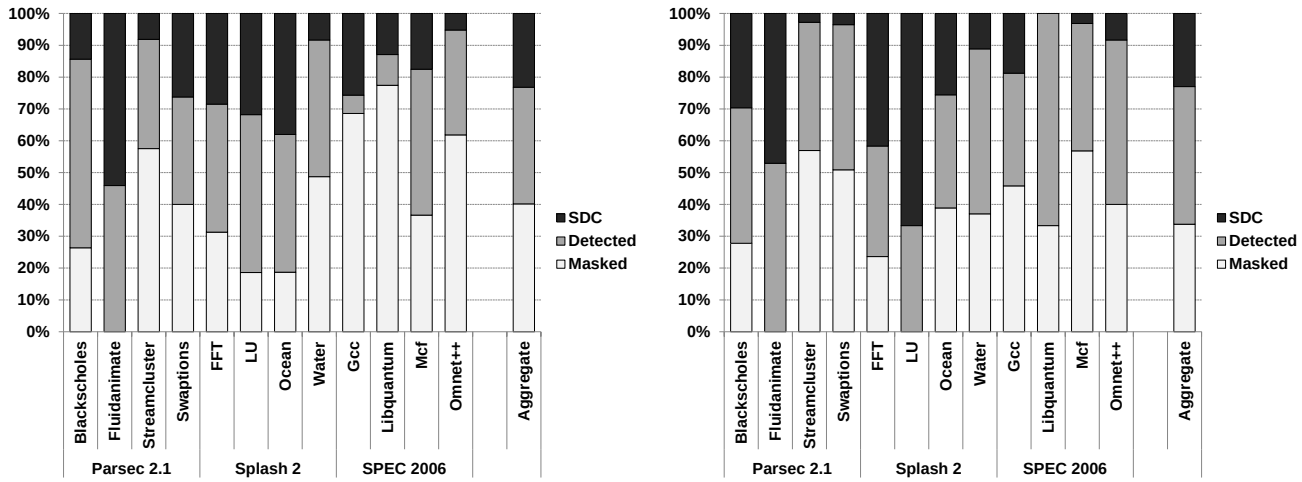
5. Related Work

There has been much work in the area of symptom-based fault detection that uses inexpensive monitors for anomalous software behavior to detect hardware faults [5, 7, 13, 14, 16, 19, 20, 25]. Much of this work is evaluated using fault injection campaigns on architecture-, microarchitecture-, or gate-level simulators or FPGA emulators running various benchmark applications. The hardware and software locations are typically randomly selected to achieve some statistical confidence. To our knowledge, these sampling mechanisms, particularly for application injection sites, have not been validated. Further, the fault injection results do not provide any insight on the parts of the application that remain vulnerable to SDCs (other than for the relatively few application sites where faults were actually injected and simulated). Relyzer selects sample application injection points derived from static and dynamic properties of the program. Each sample identifies a population of fault injection sites that are expected to have the same behavior, and all the populations together represent the entire execution. Further, Relyzer also exposes the execution trace responsible for an SDC so that alternate, customizable protection mechanisms can be designed.

Although we are not aware of other techniques that provide *all* of the above benefits of Relyzer, there are several studies that have related goals and, in some cases, are complementary to Relyzer.

SimPLFIED [18] shares Relyzer's high-level goal of finding all faults that escape detection and lead to SDCs. It uses a powerful symbolic execution method to abstract the state of erroneous values in the program. It injects such a symbolic error at all possible application sites (one at a time) and uses model checking with the abstract execution technique to explore all possible paths with the symbolic error and determines the outcomes of such paths (masking, detection, or SDC). The focus of SimPLFIED is to reduce the number of fault values per application site that need to be injected (hence the symbolic fault), which is orthogonal to our work. Our focus is to reduce the number of application sites where the fault is injected (we restrict the values by simply restricting our hardware fault models since that is not the focus of our work). It would be interesting to combine SimPLFIED with Relyzer. However, it is unclear if the model checking techniques used in SimPLFIED can scale to large applications; so far, it has been applied to only a few small benchmarks (e.g., a Siemens benchmark).

Shoestring [6] is a purely static technique that shares our goal of finding application sites where a fault may escape detection using symptom-based detectors. Shoestring provides a static analysis that identifies static instructions where faults are likely to be detected quickly enough; e.g., there is a short-enough path in the dataflow graph from such a fault to enough potentially symptom-



(a) Reg faults

(b) Agen fault

Figure 7. Breakdown of outcomes obtained from fault injections in the sampled pilots.

generating instructions. The rest of the faults are considered vulnerable and the important ones among these (currently, stores) are protected by duplicating any instructions that produce data that feeds into them. Shoestring succeeds in its goal of reducing the SDC rate by about 34% to 1.6%. However, this improvement comes at the cost of 15.8% performance overhead, which is prohibitive for the systems we target. One shortcoming of Shoestring is that it only employs static analysis to identify vulnerable instructions. Relyzer, on the contrary, applies a set of dynamic analyses that can distinguish between different instances of a static instruction and also exploit information known only at execution time (e.g., store and load addresses). It uses a combination of static and dynamic analyses to bin application fault locations into equivalence classes and then performs accurate fault simulation of the representative fault to identify the outcome. This allows Relyzer to account for masking of a fault, quantify a program's SDC rate, and enumerate the dynamic conditions that make code sections SDC prone. Shoestring's static analysis cannot achieve any of these. It would be interesting to combine the analyses of Shoestring and Relyzer in future work.

Benso et al. [2] proposed a solution that performs runtime analysis of the application variables to obtain the criticality behavior of every variable. That work developed an analytical model for this purpose that considers three variables – lifetime of the application variable, number of reads to it, and whether it is a pointer or not. The work proposes that the contribution of each of these variables is application independent and once the parameters of the model are set they remain fixed for all other applications. The results show that this solution observes low inaccuracies in predicting the criticality of variables. However, the results were shown on small applications with few variables. Relyzer, on the other hand, captures the fault propagation behavior from the fault-free execution of the application and uses it to categorize faults into different equivalence classes. It, however, relies on fault injection experiments on the pilots to estimate the outcome of the population (as opposed to estimating the outcomes based on static application-independent parameters).

Sridharan et al. [23] quantify 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 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, but this distinction becomes crucial with symptom-based detection in place. Hence, Relyzer focuses on distinguishing SDCs from detections. It is unclear whether PVF can make this distinction.

6. Conclusions and Future Work

Hardware reliability has become a major challenge, requiring low-cost and effective fault detection mechanisms. This paper concerns a promising approach, namely symptom-based detection, where low-cost monitors are deployed to monitor for anomalous software behavior as symptoms of hardware faults. There has been much recent work on such techniques, showing they provide high fault coverage at very low cost. However, for some cases, the SDC rate is still non-negligible, the evaluation techniques typically based on randomly selected fault injection campaigns are not validated, and the random injections do not provide insight on the vulnerable portions in the rest of the application that might need protection.

This paper presents Relyzer, a technique to systematically analyze all fault injection sites in an application for transient faults. Relyzer seeks to identify all SDC causing instruction instances, both to enable quantifying the application's true SDC vulnerability and to motivate low-cost application-specific protection mechanisms for the desired SDC-vulnerable cases.

Relyzer employs a set of novel fault pruning techniques that dramatically reduce the number of faults (application sites) that require thorough fault simulations. Relyzer predicts the outcome of several faults, eliminating the need for thorough fault injection experiments for them. It then exploits the fact that several application fault sites are impacted in a similar way by certain hardware faults, and develops heuristics to identify such application-level fault equivalence. Relyzer employs a series of static and dynamic techniques to categorize equivalence classes of faults, such that only one pilot fault from an equivalence class needs to be thoroughly studied through fault injection experiments. Through these techniques, we show that Relyzer prunes the set of faults by 99.78% across the twelve studied applications.

We also validated the heuristics-based fault pruning techniques by matching the results from fault simulations for the pilots with results from fault simulations with samples of the represented fault populations. Each pruning technique and fault model combination individually gave an accuracy of $>$92%, averaged across all studied

applications. Aggregating all the pruning techniques (heuristics- and analysis-based) and faults, Relyzer correctly determined the outcomes of 96% of the faults, averaged across all applications. Overall, Relyzer significantly reduced the application-level fault sites that require time-consuming simulations, making it feasible to study a complete application through a relatively small number of fault injection experiments.

For our future work, we plan to perform a sensitivity analysis of the control- and store-equivalence based techniques on the parameters such as the depth of the control flow (used to identify the control-equivalence classes) and limiting the number of tracked loads (used for finding store-equivalence classes). Our current solution prunes faults in instructions that affect a store through store-equivalence. However, the outcome of this fault may also be dependent on the control sequence of the instructions that produce the store value and accounting for this information may further improve the prediction rate. Hence it will be interesting to combine both control- and store-equivalence based pruning techniques into a unified technique. For microarchitectural fault models, applying value perturbation based refinement to the existing pruning techniques is an attractive future work because it can potentially improve the existing prediction rate (especially for Blackscholes).

In the future, we also plan to extend our study to more fault models, which might involve the development of new pruning techniques. It will be interesting to understand how this approach can be applied to permanent faults. Relyzer has been concerned, so far, about the accurate evaluation of the fault outcomes. However, evaluating the fault detection latency with high accuracy is also important. We plan to extend Relyzer in this direction as well. Since a thorough evaluation of a fault through fault simulation is expensive in time, Relyzer focused on reducing the number of fault sites requiring thorough fault injection experiments. However, techniques that directly reduce the time for each fault injection experiment are in need as well. We plan to incorporate in Relyzer an analysis that can significantly reduce the time for each fault simulation as well.

Utilizing Relyzer for evaluating an application's resiliency and identifying the SDC causing fault sites is one of our core future directions. In this work, we also plan to exploit Relyzer's ability to provide added information about the fault site and the path to the SDC in understanding the characteristics of SDC causing fault sites and developing low-cost detectors for them.

References

[1] M. Abramovici, M. A. Breuer, and A. D. Friedman. *Digital Systems Testing and Testable Design*. 1993.

[2] A. Benso, S. Di Carlo, G. Di Natale, P. Prinetto, and L. Taghaferri. Data Criticality Estimation in Software Applications. In *International Test Conference*, 2003.

[3] 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.

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

[5] 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.

[6] S. Feng, S. Gupta, A. Ansari, and S. Mahlke. Shoestring: Probabilistic Soft Error Reliability on the Cheap. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2010.

[7] O. Goloubeva, M. Rebaudengo, M. S. Reorda, and M. Violante. Soft-Error Detection Using Control Flow Assertions. In *International Symposium on Defect and Fault Tolerance in VLSI Systems*, 2003.

[8] S. K. S. Hari, M.-L. 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.

[9] J. L. Henning. SPEC CPU2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, 34:1–17, September 2006.

[10] Z. Kalbarczyk, R. Iyer, G. Ries, J. Patel, M. Lee, and Y. Xiao. Hierarchical Simulation Approach to Accurate Fault Modeling for System Dependability Evaluation. *Software Engineering, IEEE Transactions on*, 25(5):619 –632, sep/oct 1999.

[11] M.-L. 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.

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

[13] M.-L. Li, P. Ramachandran, S. Sahoo, S. V. 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.

[14] 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.

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

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

[17] S. Mirkhani, M. Lavasani, and Z. Navabi. Hierarchical Fault Simulation using Behavioral and Gate Level Hardware Models. In *Asian Test Symposium (ATS)*, pages 374 – 379, Nov. 2002.

[18] 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.

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

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

[21] P. Ramachandran. *Detecting and Recovering from In-Core Hardware Faults Through Software Anomaly Treatment*. PhD thesis, University of Illinois, Urbana Champaign, 2011.

[22] S. Sahoo, M.-L. Li, P. Ramchandran, S. Adve, V. Adve, and Y. Zhou. Using Likely Program Invariants to Detect Hardware Errors. In *International Conference on Dependable Systems and Networks*, 2008.

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

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

[25] 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.

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

[27] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *International Symposium on Computer Architecture*, 1995.