Shoestring: Probabilistic Soft Error Reliability on the Cheap

Shuguang Feng, Shantanu Gupta, Amin Ansari, and Scott Mahlke

Advanced Computer Architecture Laboratory
University of Michigan - Ann Arbor, MI
{shoe, shangupt, ansary, mahlke}@umich.edu

Abstract

Aggressive technology scaling provides designers with an ever increasing budget of cheaper and faster transistors. Unfortunately, this trend is accompanied by a decline in individual device reliability as transistors become increasingly susceptible to soft errors. We are quickly approaching a new era where resilience to soft errors is no longer a luxury that can be reserved for just processors in high-reliability, mission-critical domains. Even processors used in mainstream computing will soon require protection. However, due to tighter profit margins, reliable operation for these devices must come at little or no cost. This paper presents Shoestring, a minimally invasive software solution that provides high soft error coverage with very little overhead, enabling its deployment even in commodity processors with "shoestring" reliability budgets. Leveraging intelligent analysis at compile time, and exploiting low-cost, symptom-based error detection, Shoestring is able to focus its efforts on protecting statistically-vulnerable portions of program code. Shoestring effectively applies instruction duplication to protect only those segments of code that, when subjected to a soft error, are likely to result in user-visible faults without first exhibiting symptomatic behavior. Shoestring is able to recover from an additional 33.9% of soft errors that are undetected by a symptom-only approach, achieving an overall user-visible failure rate of 1.6%. This reliability improvement comes at a modest performance overhead of 15.8%.

Categories and Subject Descriptors B.8.1 [*Performance and Reliability*]: Reliability, Testing, and Fault Tolerance; D.3.4 [*Programming Languages*]: Processors—Compilers

General Terms Design, Experimentation, Reliability

Keywords Compiler Analysis, Error Detection, Fault Injection

1. Introduction

A critical aspect of any computer system is its reliability. Computers are expected to perform tasks not only quickly, but also correctly. Whether they are trading stocks from a laptop or watching the latest YouTube video on an iPhone, users expect their experience to be fault-free. Although it is impossible to build a completely reliable system, hardware vendors target failure rates that are imperceptibly small.

ASPLOS'10, March 13–17, 2010, Pittsburgh, Pennsylvania, USA.

One pervasive cause of computer system failure and the focus of this paper is soft errors. A soft error, or transient fault, can be induced by electrical noise or high-energy particle strikes that result from cosmic radiation and chip packaging impurities. Unlike manufacturing or design defects, which are persistent, transient faults as their name suggests, only sporadically influence program execution.

One of the first reports of soft errors came in 1978 from Intel Corporation, when chip packaging modules were contaminated with uranium from a nearby mine [13]. In 2004, Cypress semiconductor reported a number of incidents arising from soft errors [42]. In one incident, a single soft error crashed an entire data center and in another soft errors caused a billion-dollar automotive factory to halt every month.

Since the susceptibility of devices to soft error events is directly related to their size and operating voltage, current scaling trends suggest that dramatic increases in microprocessor soft error rates (SER) are inevitable. Traditionally, reliability research has focused largely on the high-performance server market. Historically the gold standards in this space have been the IBM S/360 (now Z-series servers) [32] and the HP NonStop systems [3], which rely on large scale modular redundancy to provide fault tolerance. Other research has focused on providing fault protection using redundant multithreading [8, 17, 24, 27, 30] or hardware checkers like DIVA [6, 39]. In general, these techniques are expensive in terms of both the area and power required for redundant computation and are not applicable outside mission-critical domains.

The design constraints of computer systems for the commodity electronics market differ substantially from those in the high-end server domain. In this space, area and power are primary considerations. Consumers are not willing to pay the additional costs (in terms of hardware price, performance loss, or reduced battery lifetime) for the solutions adopted in the server space. At the same time, they do not demand "five-nines" of reliability, regularly tolerating dropped phone calls, glitches in video playback, and crashes of their desktop/laptop computers (commonly caused by software bugs). The key challenge facing the consumer electronics market in future deep submicron technologies is providing just enough coverage of soft errors, such that the effective fault rate (the raw SER scaled by the available coverage) remains at level to which people have become accustomed. Examining how this coverage can be achieved "on the cheap" is the goal of this paper.

To garner statistically high soft error coverage at low overheads, we propose Shoestring, a software-centric approach for detecting and correcting soft errors. Shoestring is built upon two areas of prior research: symptom-based fault detection and software-based instruction duplication. Symptom-based detection schemes recognize that applications often exhibit anomalous behavior (symptoms) in the presence of a transient fault [11, 37]. These symptoms can include memory access exceptions, mispredicted branches, and even cache misses. Although symptom-based detection is inexpen-

sive, the amount of coverage that can be obtained from a symptom-only approach is typically limited. To address this limitation we leverage the second area of prior research, software-based instruction duplication [25, 26]. With this approach, instructions are duplicated and results are validated within a single thread of execution. This solution has the advantage of being purely software-based, requiring no specialized hardware, and can achieve nearly 100% coverage. However, the overheads in terms of performance and power are quite high since a large fraction of the application is replicated.

The key insight that Shoestring exploits is that the majority of transient faults can either be ignored (because they do not ultimately propagate to user-visible corruptions at the application level) or are easily covered by light-weight symptom-based detection. To address the remaining faults, compiler analysis is utilized to identify high-value portions of the application code that are both susceptible to soft errors (i.e., likely to corrupt system state) and statistically unlikely to be covered by the timely appearance of symptoms. These portions of the code are then protected with instruction duplication. In essence, Shoestring intelligently selects between relying on symptoms and judiciously applying instruction duplication to optimize the coverage and performance trade-off. In this manner, Shoestring transparently provides a low-cost, high-coverage solution for soft errors in processors targeted for the consumer electronics market. However, unlike the high-availability IBM and HP servers which can provide provable guarantees on coverage, Shoestring provides only opportunistic coverage, and is therefore not suitable for mission-critical applications.

The contributions of this paper are as follows:

- A transparent software solution for addressing soft errors in commodity processors that incurs minimal performance overhead while providing high fault coverage.
- A new reliability-aware compiler analysis that quantifies the likelihood that a fault corrupting an instruction will be covered by symptom-based fault detection.
- A selective instruction duplication approach that leverages compiler analysis to identify and replicate a small subset of vulnerable instructions.
- Microarchitectural fault injection experiments to demonstrate the effectiveness of Shoestring in terms of fault coverage and performance overhead.

2. Background and Motivation

2.1 Soft Error Rate

The vulnerability of individual transistors to soft errors is continuing to grow as device dimensions shrink with each new technology generation. Traditionally, soft errors were a major concern for memory cells due to their higher sensitivity to changes in operating conditions. However, protecting memory cells is relatively straightforward using parity checks or error correcting codes (ECC). On the other hand, combinational logic faults are harder to detect and correct. Furthermore, Shivakumar et al. [29] has reported that the SER for SRAM cells is expected to remain stable, while the SER for logic is steadily rising. Both these factors have motivated a flurry of research activities investigating solutions to protect the microproccessor core against transient faults. This body of related work will be addressed in Section 6.

Figure 1 shows the SER trend for a range of silicon technology generations reported in terms of *failures in time* (FIT[1]) per chip. Leveraging data presented by Shivakumar et al. [29], the SER trend for processor logic was scaled down to deep submicron technologies (similar to what is done by Borkar [5]) to generate the curve

[1] The number of failures observed per one billion hours of operation.

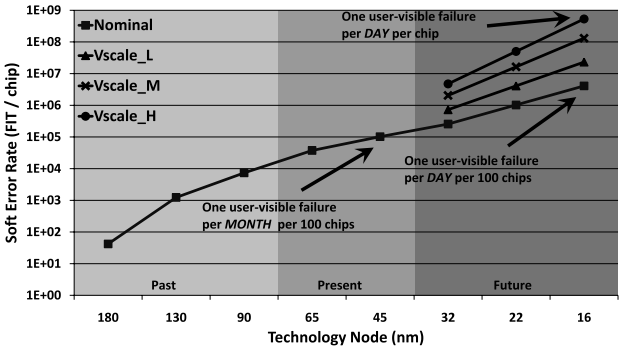


Figure 1: The soft error rate trend for processor logic across a range of silicon technology nodes. The *Nominal* curve illustrates past and present trends while the *Vscale_L*, *Vscale_M*, and *Vscale_H* curves assume low, medium and high amounts (respectively) of voltage scaling in future deep submicron technologies. The user-visible failure rates highlighted at $45\,nm$ and $16\,nm$ are calculated assuming a 92% system-wide masking rate.

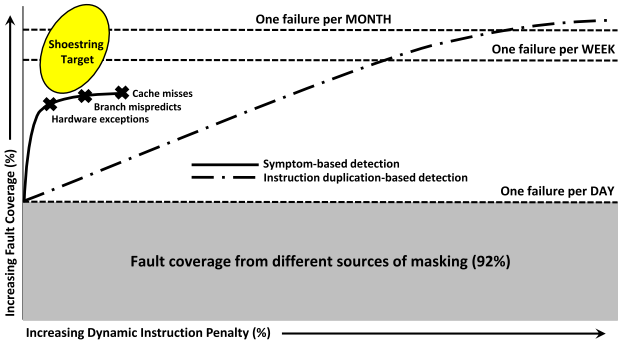


Figure 2: Fault coverage versus dynamic instruction penalty trade-off for two existing fault detection schemes: symptom-based detection and instruction duplication-based detection. Also indicated is the region of the solution space targeted by Shoestring. The mapping of fault coverage to user-visible failure rate (dashed horizontal lines) is with respect to a single chip in a $16\,nm$ technology node with aggressive voltage scaling (Vscale_H).

labeled *Nominal*. Note the exponential rise in SER with each new technology generation. Further exacerbating the SER challenge is the fact that in future technologies aggressive voltage scaling (both static and dynamic) will be required to meet power/thermal envelopes in the presence of unprecedented transistor densities. The curves, *Vscale_L*, *Vscale_M*, and *Vscale_H* illustrate the potential impact low, medium, and high amounts (respectively) of voltage scaling can have on SER.

Fortunately, a large fraction of transient faults are masked and do not corrupt actual program state. This masking can occur at the circuit, microarchitectural, or software levels. Our experiments, consistent with prior findings by Wang and Patel [35], show this masking rate to be around 92% collectively from all sources. Accounting for this masking, the raw SER at $45\,nm$ (the present technology node) translates to about one failure every month in a population of 100 chips. For a typical user of laptop/desktop computers this is likely imperceptible. However, in future nodes like $16\,nm$ the user-visible fault rate could be as high as one failure a day for every chip. The potential for this dramatic increase in the effective fault rate will necessitate incorporating soft error tolerance mechanisms into even low-cost, commodity systems.

2.2 Solution Landscape and Shoestring

As previously discussed, a soft error solution tailored for the commodity user space needs to be cheap, minimally invasive, and capable of providing *sufficient* fault coverage. Figure 2 is a conceptual plot of fault coverage versus performance overhead for the two types of fault detection schemes that form the foundation of Shoestring, one based on symptoms and the other on instruction duplication. The bottom region in this plot indicates the amount of fault coverage that results from intrinsic sources of soft error masking, available for free.

Of the remaining, unmasked faults, symptom-based detection is able to cover a significant fraction without incurring any appreciable overhead, mostly from detecting hardware exceptions. However, as a more inclusive set of symptoms are considered the overall coverage only improves incrementally while the performance overhead increases substantially. This is expected since these schemes relies on monitoring a set of rare events, treating their occurrence as symptomatic of a soft error, and initiating rollback to a lightweight checkpoint[2]. When the set of symptoms monitored is limited to events that rarely (if ever) occur under fault-free conditions (e.g., hardware exceptions) the performance overhead is negligible. However, when the set of symptoms is expanded to include more common events like branch mispredicts and cache misses, the overhead associated with false-positives increases [37].

In contrast the coverage versus performance curve is far less steep for instruction duplication. Since instruction duplication schemes achieve fault coverage by replicating computation and validating the original and duplicate code sequences, the amount of coverage is easily tunable, with coverage increasing almost linearly with the amount of duplication.

The horizontal lines in Figure 2 highlight three fault coverage thresholds that map to effective failure rates of one failure per day, week, and month (in the context of a single chip in $16\,nm$ with aggressive voltage scaling *Vscale_H*). The fault coverage provided by the intrinsic sources of masking translates to about one failure a day, clearly unacceptable. To achieve a more tolerable failure rate of one fault per week or even month, comparable to other sources of failure in consumer electronics (e.g., software, power supply, etc.), the amount of fault coverage must be significantly improved. Note that although the symptom-based detection solution is both cheap and minimally invasive, it falls short of achieving these coverage thresholds. Similarly, although instruction duplication is capable of meeting these reliability targets, it does so by sacrificing considerable performance and power (from executing more dynamic instructions).

Although neither existing technique alone provides the desired performance and coverage tradeoffs, as a hybrid method, Shoestring is able to exploit the strengths of each, ultimately providing a technique that is optimally positioned within the solution space.

3. Shoestring

The main intuition behind Shoestring is the notion that near-perfect, "five-nines" reliability is not always necessary. In fact, in most commodity systems, the presence of such ultra-high resilience may go unnoticed. Shoestring exploits this reality by advocating the use of minimally invasive techniques that provide "just enough" resilience to transient faults. This is achieved by relying on symptom-based error detection to supply the bulk of the fault coverage at little to no cost. After this low-hanging fruit is harvested, judicious application of software-based instruction duplication is then leveraged to target the remaining faults that never manifest as symptoms.

[2] The checkpointing required for the symptom detection employed by Shoestring already exists in modern processors to support performance speculation (see Section 4).

```
// inpData is a global array
// process() is a global macro
1: index = 0
2: while (!stop)
3:   process(inpData[index])
4:   process(inpData[index + 1])
5:   process(inpData[index + 2])
6:   process(inpData[index + 3])
7:   index = index + 4
8:   stop = (index + 3) >= inpDataSize
9: end
10: // clean-up code
11: for (; index < inpDataSize; index++)
12:   process (inpData[index])
13: end
```

Figure 3: A representative example of performance optimized code (loop unrolled).

To the first order, program execution consists of data computation and traversing the control flow graph (CFG). Correct program execution, strictly speaking, requires 1) that data be computed properly and 2) that execution proceeds down the right paths, i.e., compute the data *correctly* and compute the *correct* data. Working from this definition, previous software-based reliability schemes like SWIFT [25] have assumed that a program executes correctly (from the user's perspective) if all stores in the program are performed properly. This essentially redefines correct program execution as 1) storing the correct data (to the correct addresses) and 2) performing the right stores. Implicit is the assumption that the sphere of replication (SoR) [24], or the scope beyond which a technique cannot tolerate faults, is limited to the processing core. Faults in the caches and external memories are not addressed, but can be efficiently protected by techniques like ECC [9].

Shoestring, makes similar assumptions about SoR and correct program execution. However, unlike SWIFT [25] and other schemes, we are not targeting *complete* fault coverage. Relaxing the coverage constraint frees Shoestring from having to protect all portions of a program in order to guarantee correctness. This affords Shoestring the flexibility to only selectively protect those stores that are most likely to impact program output and least likely to already be covered by symptom detectors. Furthermore, we acknowledge that recent work by Wang et al. [36] has shown that as many as 40% of all dynamic branches are *outcome tolerant*. That is, they do not affect correct program behavior when forced down the incorrect path. The authors demonstrate that many of these so-called "Y-branches" are the result of partially dead control (i.e., they are data dependent and outcome tolerant the *majority* of the time). Leveraging this insight, Shoestring can also shed the overhead required to ensure that the CFG is *always* properly traversed. Instead, we focus on only protecting a subset of control flow decisions that impact "high-value" instructions.

Figure 3 shows a snippet of code where some manipulation of an array data structure is being performed. The computation is performed within a tight loop that uses the `process` macro to manipulate elements of the array `data`. Performance optimizations cause the loop to be unrolled 4 times into lines 2 through 9. Additional cleanup code (lines 11 through 13) is also inserted to maintain program semantics. Note that in this example not all computation is essential for correct program behavior. The instruction at line 8 determines if the early loop termination condition is met. If the instruction(s) computing `stop` is (are) subjected to a transient fault, the unrolled loop could exit prematurely. Although this early exit degrades performance, program correctness is still maintained. In contrast, properly updating the variable `index` at line 7 is re-

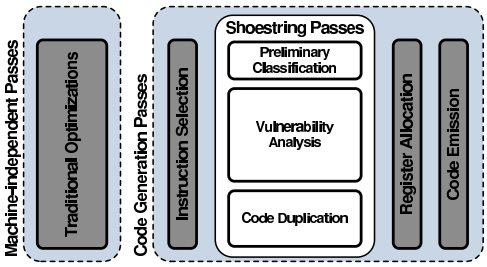


Figure 4: A standard compiler flow augmented with Shoestring's reliability-aware code generation passes.

quired for program correctness (assuming of course that `inpData` is a user-visible variable). However, since `index` is also used as a base address to access `inpData`, there is a significant probability that a fault corrupting `index` would manifest as a symptomatic memory access exception. Given the proper symptom-based detection scheme, this could decrease the *effective* vulnerability of the computation at line 7. Identifying instructions critical to program correctness and pruning from this set those instructions that are already "covered" by symptom-based detection is the focus of the remainder of this section.

3.1 Compiler Overview

Implementing the most cost effective means of deploying instruction duplication requires detailed compiler analysis. Shoestring introduces additional reliability-aware code generation passes into the standard compiler backend. Figure 4 highlights these passes in the context of typical program compilation. Shoestring's compilation passes are scheduled after the program has already been lowered to the machine-specific representation but before register allocation.

The first two passes, *Preliminary Classification* and *Vulnerability Analysis*, are designed to categorize instructions based on their expected behavior in the presence of a transient fault. These categories are briefly described below.

- **Symptom-generating**: these instructions, if they consume a corrupted input, are likely to produce detectable symptoms.
- **High-value**: these instructions, if they consume a corrupted input, are likely to produce outputs that result in user-visible program corruption.
- **Safe**: these instructions are naturally covered by symptom-generating consumers. For any safe instruction, I_S, the expectation is that if a transient fault is propagated by I_S, or arises during its execution, there is a high probability that one of its consumers will generate a symptom within an acceptable latency S_{lat}.
- **Vulnerable**: all instructions that are not safe are considered vulnerable.

Following the initial characterization passes, a third pass, *Code Duplication*, performs selective, software-based instruction duplication to protect instructions that are not inherently covered by symptoms. This duplication pass further minimizes wasted effort by protecting only the high-value instructions, those likely to impact program output. By only duplicating instructions that are along the dataflow graph (DFG) between safe and high-value instructions, the performance overhead can be dramatically reduced without significantly impacting reliability.

The following sections describe the details of the heuristics used in the analysis and duplication passes.

3.2 Preliminary Classification

Shoestring's initial characterization pass iterates over all instructions in the program and identifies symptom-generating and high-value instructions. For clarity, this classification is described as a separate compiler pass. However, in practice the identification of symptom-generating and high-value instructions can be performed as part of the vulnerability analysis pass.

3.2.1 Symptom-generating Instructions

The symptom events considered by prior symptom-based detection work can be broadly separated into the following categories [11, 37]:

- **ISA-defined Exceptions**: these are exceptions defined by the instruction set architecture (ISA) and must already be detected by any hardware implementing the ISA (e.g., page fault or overflow).
- **Fatal Exceptions**: these are the subset of the ISA-defined exceptions that never occur under normal user program execution (e.g., segment fault or illegal opcode).
- **Anomalous Behavior**: these events occur during normal program execution but can also be symptomatic of a fault (e.g., branch mispredict or cache miss).

The relative usefulness of symptoms in each of these categories is dependent on how strongly their appearance is correlated with an actual fault. Ideal candidates occur very rarely during normal execution, minimizing overhead due to false positives, but always manifest in the wake of a fault. Therefore, to maximize the overhead-to-coverage tradeoff the experiments in Section 5 evaluate a Shoestring implementation that only considers instructions that can elicit the second category of fatal, ISA-defined exceptions as potentially symptom generating. Since these are events that during the normal execution of user programs never arise, they incur no performance overhead in the absence of faults.

Although additional coverage can be gleaned by evaluating a more inclusive set of symptoms, prior work has shown that the additional coverage often does not justify the accompanying costs. For example, Wang and Patel [37] presented results where using branch mispredictions on high-confidence branches as a symptom gained an additional 0.3% of coverage with an 8% performance penalty. Other non-fatal symptoms like data cache misses also have similar coverage and overhead profiles.

3.2.2 High-value Instructions

Ideally, we would like to only consider instructions that impact program output as high-value. However, given that the analysis necessary to provably make such determinations is impractical, if not intractable, heuristics must be employed. Currently, any instructions that can potentially impact global memory is considered high-value. In addition, any instructions that can produce arguments passed to function calls (especially library calls) are also included. To provide a truly transparent solution, Shoestring, at present, assumes that no user annotations are available to assist in instruction classification. Future extensions could leverage techniques from information-flow theory [14, 33] to further refine the instruction selection process or even exploit the work by Li and Yeung [12] to prune instructions that only impact "soft" program outputs. Although investigating more sophisticated heuristics for identifying high-value instructions is a very promising avenue of future work, it is beyond the scope of the current paper.

3.3 Vulnerability Analysis

After the preliminary instruction classification is complete, Shoestring analyzes the vulnerability of each instruction to determine whether

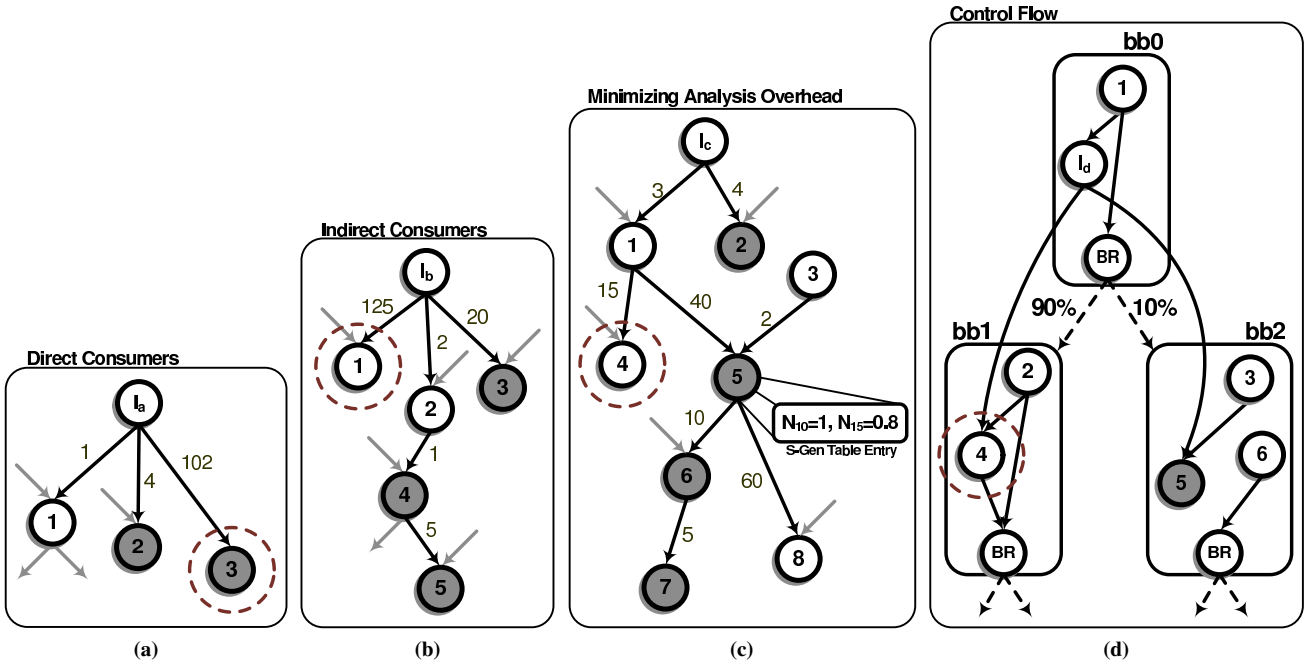


Figure 5: Example data flow graphs illustrating Shoestring's vulnerability analysis. The data flow edge numbers represent the distance between two instructions in the statically scheduled code. Shaded nodes represent symptom-generating instructions and dashed circles highlight high-value instructions. Dashed edges in (d) represent control flow.

it is safe. As stated previously, a safe instruction, I_S, is one with enough symptom-generating consumers such that a fault corrupting the result of I_S is likely to exercise a symptom within a fixed latency S_{lat}. For each instruction, the number of symptom-generating consumers is first tabulated based on distance. For a given producer (I_p) and consumer (I_c) pair, we define the distance, $D_{p,c}$, as the number of intervening instructions between I_p and I_c within the statically scheduled code. It is used as a compile-time estimate of the symptom detection latency if the consumer, I_c, were to trigger a symptom. For a given instruction, I, if the number of symptom-generating consumers at distance i is N_i, then I is considered safe if $N_{tot} = \sum_{i=1}^{S_{lat}} N_i$ is greater than a fixed threshold S_t. The value for the threshold parameter S_t controls the selectivity of safe instruction classification and can be used to trade off coverage for performance overhead (see Section 5).

Figure 5 and the corresponding case studies illustrate how the vulnerability analysis heuristic is applied for a few sample DFGs. The numbers along the data-flow edges represent the distance, $D_{p,c}$, between the two nodes (instructions). Shaded nodes indicate symptom-generating instructions, and nodes highlighted by a dashed circle are high-value instructions. For all the case studies, $S_{lat} = 100$ and $S_t = 2$.

3.3.1 Case Study 1: Direct Consumers

In Figure 5a, instruction I_a is being analyzed for safe-ness. Instructions 1, 2, and 3 are all direct consumers of I_a. Instructions 2 and 3 have already been identified as symptom-generating instructions and 3 is also a high-value instruction. In this example, I_a would be classified as vulnerable because it only has one symptom-generating consumer within a distance of 100 (S_{lat}), instruction 2.

3.3.2 Case Study 2: Indirect Consumers

Figure 5b presents a more interesting example that includes direct as well as indirect consumers as we analyze I_b. As with direct consumers, indirect consumers that have been identified as symptom-generating also contribute to the N_{tot} of I_b. However, their contribution is reduced by a scaling factor S_{iscale} to to account for the potential for *partial* fault masking.

In Figure 5b, instructions 3, 4, and 5 are all symptom generating consumers of I_b. Since 3 is a direct consumer, any fault that corrupts the result of I_b will cause instruction 3 to generate a symptom (probabilistically of course). However, the same fault would have to propagate through instruction 2 before it reaches the indirect consumer, instruction 4. This allows for the possibility that the fault may be masked by 2 before it actually reaches 4. For example, if the soft error flipped an upper bit in the result of I_b and instruction 2 was an `AND` that masked the upper bits, the fault would never be visible to instruction 4, reducing its ability to manifest a symptom. However, instruction 1 would still consume the tainted value and potentially write it out to memory, corrupting system state. Therefore, due to the potential for masking, an indirect consumer is less likely than a direct consumer to cover the exact same fault. Ultimately with respect to I_b in Figure 5b, given an $S_{iscale} = 0.8$, we have $N_{20} = 1$, $N_3 = 0.8$, and $N_5 = 0.64$. Since $N_{tot} = \sum_{i=1}^{100} N_i = 2.44$ and is greater than the threshold S_t of 2, I_b is classified as safe.

3.3.3 Case Study 3: Minimizing Analysis Overhead

Figure 5c presents a more complete example and further illustrates how memoization is used to avoid redundant computation. Rather than identifying indirect symptom-generating consumers recursively for every instruction, we maintain a global *symptom-generation table* of N_i values for every instruction. By traversing the DFG in a depth-first fashion, we guarantee that all the consumers of an instruction are processed before the instruction itself is encountered. Creating an entry in the symptom-generation table (labeled S-Gen Table in the Figure 5c) for every instruction as it is being analyzed ensures that each node in the DFG only needs to be visited once[3].

[3] Although this optimization is beneficial for programs with large functions, even a naive recursive analysis for the SPEC2K applications evaluated in this work did not appreciably increase compilation time.

For example, assuming the vulnerability analysis begins with I_c, Shoestring analyzes the instructions in the following order, 4, 7, 6, 8, 5, 1, 2 and eventually marks I_c as safe. When the analysis pass reaches instruction 3 it can determine its classification directly, without identifying any of its indirect consumers, since the symptom-generation table entry for instruction 5 was already populated during the analysis pass for I_c. The corresponding table entry for 3 is computed by scaling all N_i entries for 5 by S_{iscale}, adjusting the corresponding distances by adding 2, and finally accounting for the symptom-generating potential of instruction 5 itself. The table entry for instruction 3 would then contain $N_2 = 1, N_{12} = 0.8, N_{17} = 0.64$ and instruction 3 would subsequently also be classified as safe.

Obviously, this depth-first traversal is complicated in the presence of loops (not present in Figure 5c) where circular dependencies can exist and the traversal could loop indefinitely never reaching a leaf node. Consequently, whenever Shoestring encounters a loop it forces the depth-first traversal to backtrack when the distance between the instruction currently being processed and the instruction at the bottom of the loop exceeds S_{lat}. This guarantees all relevant symptom-generating consumers are accounted for while also ensuring forward progress.

3.3.4 Case Study 4: Control Flow

The examples examined so far have been limited to analyzing instruction DFGs and control has to a large extent been ignored. Although Shoestring takes a relaxed approach with respect to *enforcing* correct control flow, branching is taken into consideration when performing vulnerability analysis. Figure 5d shows an example where the instruction being analyzed, I_d, is in a basic block (bb0) that has a highly biased branch. In this scenario, although instruction 5 is a symptom-generating consumer, because it is in a basic block (bb2) that is unlikely to be executed, it will not provide dependable coverage for I_d. Therefore, the contribution of every consumer to N_i is scaled by their respective execution probabilities. These execution probabilities are extracted from profiled execution (provided to the experiments in Section 5), or when profile data is unavailable, generated from static approximations.

Lastly, although Wang et al. [36] showed that execution down the wrong direction of many branches ultimately reconverges with the correct execution path, in Figure 5d if the branch terminating bb0 is corrupted causing execution to proceed to bb2 instead of bb1, there is no time for control to reconverge before instruction 4 potentially corrupts system state. Therefore, Shoestring also selectively protects (by duplicating input operand chains) all branches that have a control-dependence edge with a high-value instruction. For sake of brevity, the standard algorithm for identifying control-dependence edges will not be presented here but it is important to note that not all branches that can influence whether instruction 4 is executed will be protected. Only those branches that are effectively the "nearest" to instruction 4 will possess the requisite control-dependence edges and be protected, leaving the rest (which are further away and more likely to reconverge) vulnerable.

3.4 Code Duplication

The process of inserting redundant code into a single thread of execution has been well studied in the past [19, 26]. In general, this process involves duplicating all computation instructions along the path of replication and inserting comparison instructions at synchronization points (e.g., at memory and control flow instructions) to determine if faults have manifested since the last comparison was performed. This section will highlight how Shoestring's code duplication pass departs from this existing practice. The reader is encouraged to examine prior work for a detailed description of the mechanics of code duplication.

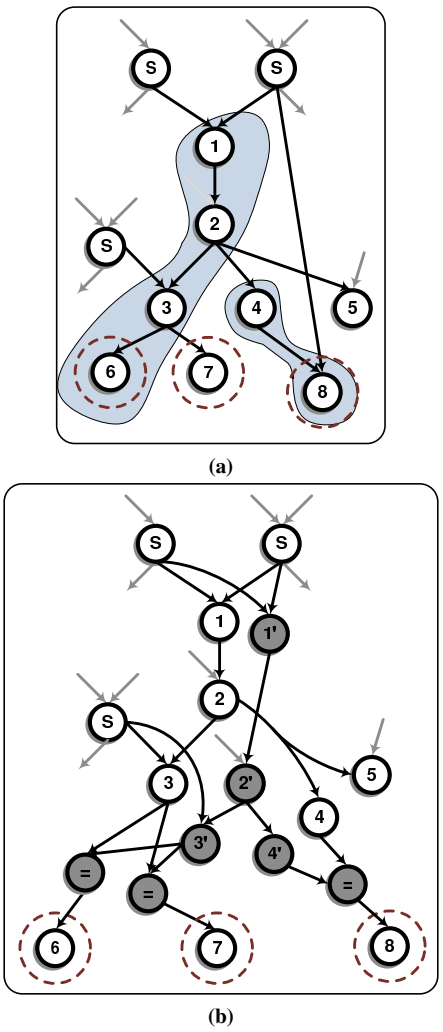


Figure 6: Example data flow graph illustrating Shoestring's code duplication pass. Nodes labeled with an "S" represent safe instructions and dashed circles highlight high-value instructions. In (a) the shaded portions of the graph represent code duplication chains. (b) shows the new DFG with all duplicated instructions inserted as shaded nodes. Nodes labeled with an "=" represent checker instructions.

The code duplication pass begins by selecting a single high-value instruction, I_{HV}, from the set of all high-value instructions. It then proceeds to recursively duplicate all instructions that produce values for I_{HV}. This duplication is terminated when 1) no more producers exist, 2) a safe instruction is encountered, or 3) the producer has already been previously duplicated. In all cases, it is guaranteed that every vulnerable instructions that could possibly influence data consumed by I_{HV} is duplicated. Comparison instructions are inserted right before I_{HV} to verify the computation of each of its input operands.

Figure 6a presents a section of a DFG with three high-value instructions (nodes 6, 7, and 8), three safe instructions (nodes labeled with an "S"), and five vulnerable instructions (nodes 1-5). For this example, we start with instruction 6 and begin by duplicating its producer, instruction 3. Next, we attempt to duplicate the producers for 3 and notice that one of the producers has been classified as safe and terminate code duplication on that path. The other producer for 3 (instruction 2), however, is vulnerable so we duplicate 2 and continue along its producer chain duplicating instruction 1 as well.

Subsequent attempts to duplicate 1's consumers encounters safe instructions, at which point all vulnerable code relevant to high-value instruction 6 has been duplicated. Shoestring then moves on to the next high-value instruction and repeats the process with instruction 7. At this point, instruction 3 has already been duplicated as a result of protecting instruction 6 so nothing needs to be done. Next, instruction 8 is considered, resulting in the duplication of instruction 4.

Figure 6b shows the new DFG with all the duplicated instructions (shaded nodes) and checkers ("=" nodes) inserted. Note that both high-value instructions 6 and 7 each have their own checker to compare the results from instruction 3 and its redundant copy 3'. Although both 6 and 7 consume the same value, only relying on a single checker at instruction 6 to detect faults that corrupt 3's result could leave 7 vulnerable to faults that corrupt the result of 3 after 6 has already executed. Depending on how far apart 6 and 7 execute, this vulnerability window could be significant. Nevertheless, in situations where high-value instructions with common producers also execute in close proximity, the need for duplicate checkers can also be avoided. However, this optimization is not investigated in this work.

4. Experimental Methodology

Given that this paper is targeting coverage of faults induced by soft errors on *commodity* processors, we would ideally conduct electron beam experiments using real hardware running code instrumented by Shoestring. Given limited resources a popular alternative to beam experiments is statistical fault injection (SFI) into a detailed register transfer language (RTL) processor model. However, since Shoestring exploits fault masking at the application level, full program simulation is also required. Since simulating realistic benchmarks on RTL models is extremely slow, a common practice in the literature is to rely on microarchitectural-level simulators to provide the appropriate compromise between simulation fidelity and speed.

4.1 Fault Model and Injection Framework

The fault injection results presented in this paper are generated using the PTLsim x86 microarchitectural simulator [41]. PTLsim is able to run x86 binaries on the native (host) machine as well as within a detailed microarchitectural simulator. Being able to effectively switch between native hardware execution and microarchitectural simulation on-the-fly enables fast, highly detailed simulations. We simulated a modern, high performance, out-of-order processor modeled after the AMD K8 running x86 binaries. The details of the processor configuration can be found in Table 1.

The fault model we assume is a single bit flip within the physical register file. Although they are not explicitly modeled, most faults in other portions of the processor eventually manifest as corrupted state in the register file, making it an attractive target for injection studies[4]. Furthermore, Wang et al. [38] showed that the bulk of transient-induced failures are dominated by corruptions introduced from injections into the register file. Nevertheless, our methodology may not fully capture the ability of Shoestring to handle faults from combinational logic with large fanouts.

The experimental results shown in this paper are produced with Monte Carlo simulations. At the start of each Monte Carlo trial a random physical register bit is selected for injection. It has been shown that the memory footprint of SPEC2K applications are significantly smaller than the full size of a 64-bit virtual address space.

[4] Only performing fault injections into the register file is a limitation of our evaluation infrastructure, not a limitation of Shoestring's fault coverage abilities. In reality Shoestring will detect soft errors that strike other parts of the processing core as well.

Table 1: Processor details (configured to model an AMD-K8).

Processor core @ 2.2GHz	
Fetch queue size	36 entries
Reorder buffer size	72 entries
Issue queue size	16 entries
Issue width	16 entries
Fetch/Dispatch/Writeback/ Commit width	3
Load/Store queue size	44 entries (each)
Physical register file size	128 entries
Physical register file size	128 entries
Memory	
L1-I/L1-D cache	64KB, 2-way, 3 cycle lat
L2 cache (unified)	1MB, 16-way, 10 cycle latency
DTLB/ITLB	32 entries (each)
Main memory	112 cycle lat

Allowing faults to occur in any of the 64-bits of a register would increase the likelihood of it resulting in a symptomatic exception [37], and consequently being covered by Shoestring. Therefore, although PTLsim simulates a 64-bit register file, we limit our fault injections to only the lower 32 bits to avoid artificially inflating Shoestring's coverage results.

Once an injection site is determined, program simulation is allowed to run in native mode (running on real hardware) until it reaches a representative code segment (identified using SimPoint [28] and manual source code inspection). At this point PTLsim switches to detailed mode and warms up the microarchitectural simulator. After a randomly selected number of cycles has elapsed, a fault is induced at the predetermined injection site. Detailed simulation continues until 10M instructions commit, at which time PTLSim copies architectural state back to the host machine and resumes simulating the remainder of the program in native mode. This of course assumes that the fault did not result in a fatal exception or program crash prior to 10M instructions. At the end of every simulation the log files are analyzed to determine the outcome of the Monte Carlo run as described in the next section.

4.2 Outcome Classification

The result of each Monte Carlo trial is classified into one of four categories:

1. **Masked**: the injected fault was naturally masked by the system stack. This includes trials where the fault was architecturally masked as well as those that were masked at the application level.
2. **Covered by symptoms**: the injected fault resulted in anomalous program behavior that is symptomatic of a transient fault. For these trials it is assumed that system firmware is able to trigger recovery from a lightweight checkpoint. The details of this assumed checkpointing mechanism are described in the next section.
3. **Covered by duplication**: faults in this category were the result of injecting code that was selectively duplicated by Shoestring. During these trials the comparison instructions at the end of every duplication chain would trigger a function call to initiate recovery.
4. **Failed**: In this work the definition of failure is limited to only those simulation runs which completed (or prematurely terminated) with user-visible data corruptions.

Although the definition for failure used in this paper may seem unconventional, it is consistent with recent symptom-based work and is the most appropriate in the context of evaluating Shoestring. The main premise behind the Shoestring philosophy is that the cost of ensuring reliable computation can be reduced by focusing on covering only the faults that are ultimately noticeable by the end user. Therefore, the figure of merit is not the number of faults that propagated into microarchitectural (or architectural) state, but rather the fraction that actually resulted in user-visible failures.

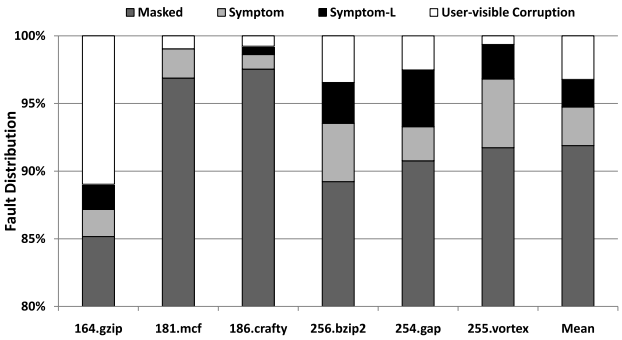
4.3 System Support

As briefly discussed in the previous section, Shoestring (and symptom-based schemes in general) relies on the ability to roll-back processor state to a clean checkpoint. The results presented in Section 5 assume that in modern/future processors a mechanism for recovering to a checkpointed state of 10-100 instructions in the past will already be required for aggressive performance speculation. Consistent with Wang and Patel [37], Shoestring assumes that any fault that manifests as a symptom within a window of 100 committed instructions (micro-ops, not x86 instructions) can be safely detected and recovered. The proper selection of the S_{lat} parameter described in Section 3.3 is closely tied to the size of this checkpointing window. Only those consumers that can be expected to generate a symptom within this window are considered when identifying safe instructions. Similarly, faults that are detected by instruction duplication would also trigger rollback and recovery.

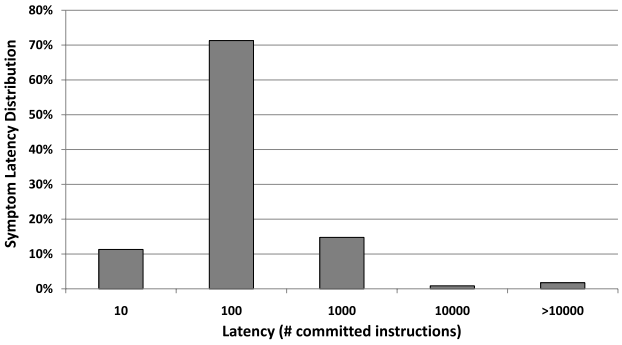
The results presented in Section 5 assume a checkpointing interval, and consequently an S_{lat} value, of 100. Although this small window may seem modest in comparison to checkpointing intervals assumed by other work, most notably Li et al. [11], it is the most appropriate given Shoestring's goals of providing minimally invasive, low cost protection. Increasing the size of this window would unfairly inflate the coverage provided by Shoestring since accommodating large checkpointing intervals requires substantial hardware/software overhead. However, if large checkpointing intervals eventually find their way into mainstream processors, the heuristics used by Shoestring can be easily tuned to exploit this additional support and provide even greater coverage.

The compilation component of Shoestring is implemented in the LLVM compiler [10]. The reliability-aware code generation passes described in Section 3.1 are integrated as part of the code generation backend. Six applications from the SPEC2K integer benchmark suite (*gzip, mcf, crafty, bzip2, gap*, and *vortex*) are used as representative workloads in our experiments and are compiled with standard -O3 optimizations. To minimize initial engineering effort, we only evaluated benchmarks from the SPEC2K suite that both 1) compiled on standard LLVM (without modifications for Shoestring) and 2) simulated correctly on PTLSim "out-of-the-box". They were not handpicked because they exhibited desirable behavior. Similarly, to minimize engineering effort we do not apply Shoestring to library calls. The common practice in the literature is to assume that dynamically linked library calls are protected by some other means, i.e., outside the SoR (see Section 3) [25]. The results presented in Section 5 adheres to the same practice and avoids injections into library calls.

Lastly, due to limitations of our evaluation framework we do not study Shoestring in the context of multithreaded/multicore environments. Given that we treat the cache as outside our SoR the majority of challenges posed by the shared memory in multithreaded/multicore systems would not impact the efficacy of Shoestring. However, the larger memory footprints of multithreaded applications could potentially attenuate the coverage due to a reduction in the performance of memory access symptoms. The greater resource/register utilization in simultaneous multithreaded systems could also reduce the amount of masking we see from



(a) Symptom-based fault coverage



(b) Latency distribution of symptom-based detection.

Figure 7: Results of preliminary fault injection experiments. (a) shows the percentage of faults that are intrinsically masked (*Masked*), covered by symptoms (*Symptom*), covered by long-latency symptoms (*Symptom-L*), or result in user-visible failures (*User-visible Corruption*).

faults that strike dead/free registers. Lastly, identifying the exact core which triggered the symptom, as well as orchestrating the checkpoint rollback and recovery, is more challenging when multiple threads running on different cores are interacting and sharing data. However, these challenges are beyond the scope of the current paper and are left as interesting directions for future work.

5. Evaluation and Analysis

This section begins with results from an initial fault injection campaign to quantify the amount of opportunity available for Shoestring to exploit. We then proceed to examine the compilation heuristics described in Section 3. Finally, we present and analyze the fault coverage and runtime overheads for Shoestring. All experimental results included in this section are derived from >10k Monte Carlo trials.

5.1 Preliminary Fault Injection

Figure 7 presents the results of initial fault injection trials. The purpose of this preliminary experiment was to identify the amount of faults that are inherently masked throughout the entire system stack. The accumulation of all these sources of masking, from the microarchitecture up through the application layer, is essentially the amount of "coverage" that is available for free. This is shown as the *Masked* segment on the stacked bars and corresponds to roughly 91.9% on average across the benchmarks. Symptoms account for another 4.9% and actual user-visible failures account for the remaining 3.2%.

As mentioned in Section 3 symptom-based coverage is only useful if symptoms are triggered within a small window of cycles,

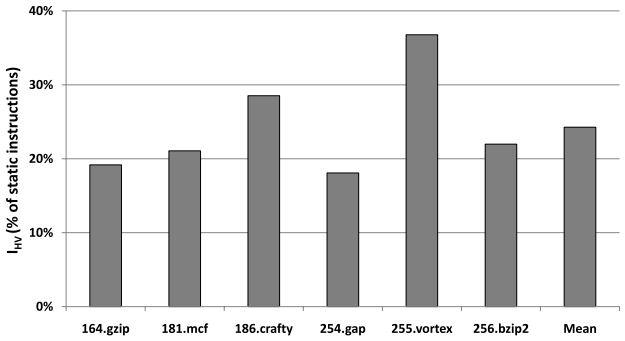


Figure 8: Percentage of static instructions classified as high-value (I_{HV}).

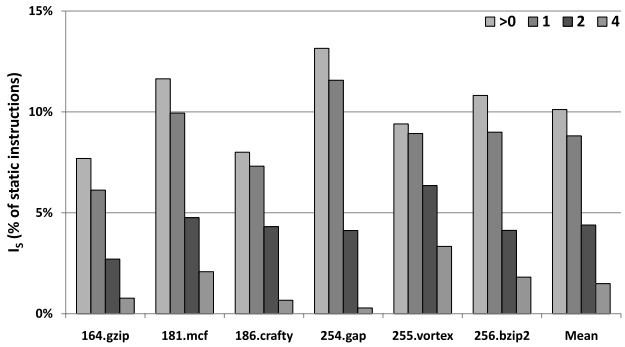


Figure 9: Percentage of static instructions classified as safe as S_t is varied (I_S).

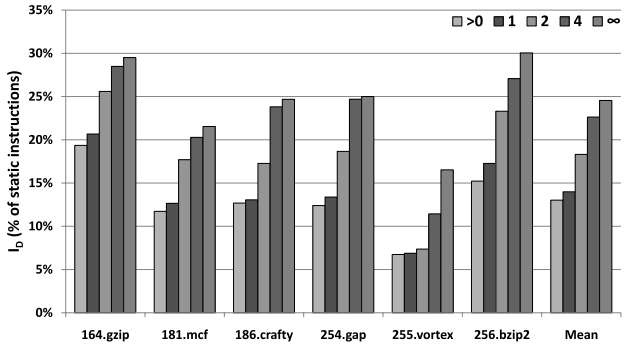


Figure 10: Percentage of static code duplication performed by Shoestring as S_t is varied (I_D).

S_{lat}, following a fault. If the symptom latency exceeds S_{lat} then the likelihood that state corruption can occur before a symptom is manifested increases. Note now the portions of the chart labeled as *Symptom-L*. These segments are the fraction of trials that lead to symptoms but did so only after the 100 instruction S_{lat} window expired. Without more expensive check-pointing to accommodate longer latencies, the Symptom-L cases must also be considered failures. Figure 7b examines these symptom-generating trials from a different perspective, as a distribution based on detection latency. Although the majority of symptoms do manifest within the 100 instruction threshold, roughly 14.7% would require a much more aggressive checkpointing scheme (1000 instructions) than what is needed for performance speculation alone. Furthermore, the remaining 2.6%, with latencies of more than 10,000 instructions could not be exploited without being accompanied by heavyweight, software-based checkpointing (and its attendant costs). The remainder of the paper assesses Shoestring's ability to minimize user-visible corruptions by integrating symptom-based coverage with intelligent software-based code duplication.

5.2 Program Analysis

5.2.1 High-value Instructions

To gain insight into how selective instruction duplication is actually applied by Shoestring, we examine the heuristics described in Section 3 in the context of our SPEC2K workloads. Figure 8 shows the percentage of instructions identified as high-value within each benchmark. As discussed in Section 3.2.2, only instructions that can potentially modify global memory or produce values for function calls are considered high-value. On average roughly 24.3% of all static instructions meet this criteria and become the focus of Shoestring's code duplication efforts.

5.2.2 Safe Instructions

Figure 9 presents the percentage of instructions classified as safe, as a function of the heuristic parameter S_t (Section 3.3). A value of n for S_t indicates that for an instruction to be considered safe (i.e., covered by symptom-generating consumers) it must possess at least n consumers within a distance of S_{lat} instructions along any potential path of execution. Note that on average, even with S_t relaxed to allow for any non-zero threshold (>0) only 10.1% of static instructions are classified as safe. This is mainly due to our conservative decision to only consider potential ISA-excepting instructions as candidates for symptom-generating consumers. A more aggressive heuristic could potentially identify more safe instructions if the set of symptoms that it monitored was more inclusive. However, this would come at the cost of performance-degrading false positives.

For this, and all subsequent, experiments the value of S_{lat} was fixed at 100 instructions as explained in Section 4.3. A value of 0.9 for S_{iscale} (Section 3.3.2) was also empirically determined to produce the best heuristic behavior, and is fixed for all experiments.

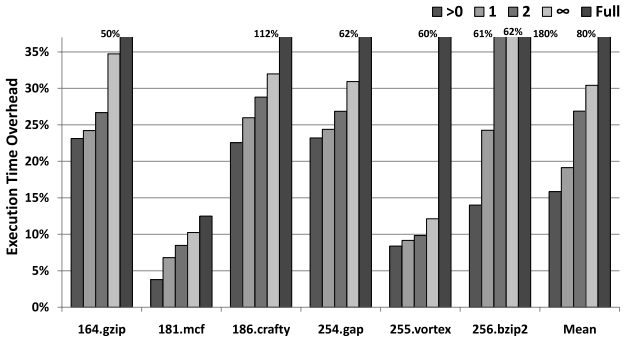
5.2.3 Duplicated Instructions

Figure 10 shows the percentage of (static) instructions that are duplicated by Shoestring as S_t is swept from >0 to ∞. Note the direct relationship between S_t and the number of duplicated instructions. This is attributable to the fact that code duplication begins at high-value instructions and terminates at the first safe instruction encountered (see Section 3.4). Therefore, the fewer instructions that are classified as safe, the less likely a duplication chain will terminate early. In the extreme case when $S_t = \infty$ no instructions are classified as safe. This results in fully duplicating producer chains for every high-value instruction.

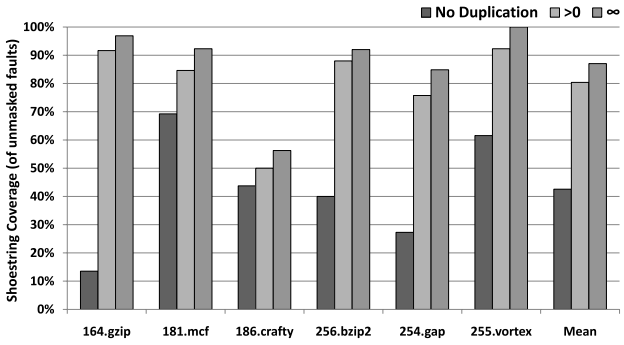
5.3 Overheads and Fault Coverage

Next we examine the runtime overhead of the binaries that have been protected by Shoestring's selective code duplication. Figure 11a shows that as S_t is varied from >0 to ∞ the performance overhead of Shoestring ranges from 15.8% to 30.4% (obtained from native simulation on an Intel Core 2 processor). The execution overheads for a full software duplication scheme, SWIFT [25], are also included for the sake of comparison[5](bars labeled *Full*). Since Shoestring is positioned as a reliability "on the cheap" solution, maintaining low runtime overhead is of paramount importance. To evaluate the amount of coverage that can be obtained by Shoestring

[5] The overheads we cite from SWIFT [25] are conservative considering they targeted a wide VLIW machine and would incur substantially more overhead given a less overprovisioned processor.



(a) Runtime performance overhead.



(b) Fault coverage.

Figure 11: Fault coverage and runtime performance overheads for Shoestring as a function of S_t.

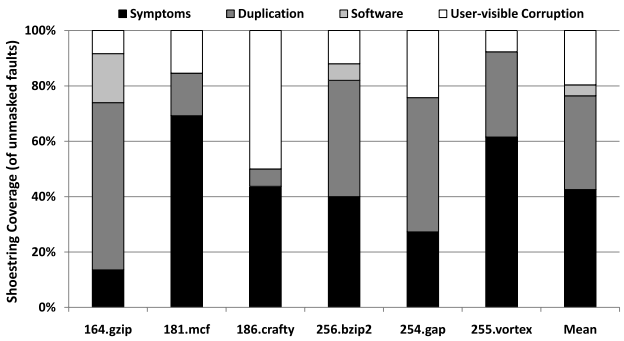


Figure 12: Detailed coverage breakdown for Shoestring configured with S_t fixed at >0.

with the least amount of performance overhead, for the remainder of this section the value of S_t is fixed at >0.

Figure 11b presents the coverage results for this Shoestring configuration. Also included are coverage numbers for no instruction duplication, *No Duplication*, and $S_t = \infty$ to illustrate where the proposed solution sits with respect to the upper and lower bounds. Coverage numbers for other values of S_t were not included because of the requisite simulation effort. Although investigating more sophisticated heuristics for instruction classification and vulnerability analysis has the potential to garner even more coverage, note that Shoestring is already able to recover from 80.4% of the failures that would have otherwise gone unmasked and caused user-visible data corruptions.

Lastly, Figure 12 takes a closer look at the fault coverage achieved by Shoestring. The stacked bars highlight the individual components contributing to Shoestring's total fault coverage. Note that on average, selective duplication covers an additional 33.9% of the unmasked faults that would have slipped by a symptom-only based scheme. Notice also the segment labeled *Software*. This category, only significant for *gzip* and *bzip2*, is the result of assertions placed in the source code that actually detect the erroneous behavior of the program following a fault injection. This observation suggests that perhaps only a modest investment is required to enhance conventional assertion checks with information that could improve Shoestring's program analysis.

6. Related Work

This section examines Shoestring in the context of previous work. Table 2 presents a quick overview of where Shoestring sits within the soft error solution space. The ability to achieve high levels of fault coverage with very low performance overhead, all without any specialized hardware, sets it apart from previously proposed schemes. Each category of alternative solutions is addressed in detail below.

n-Modular Redundancy (nMR): Spatial or temporal redundant execution has long been a cornerstone for detecting soft errors, with hardware DMR (dual-modular redundancy) and TMR (triple-modular redundancy) being the solutions of choice for mission-critical systems. However, the cost of such techniques has relegated them to the high-budget server and mainframe domains (e.g., HP NonStop series [3] and IBM zSeries [2] machines). DIVA [1] is a less expensive alternative to full hardware duplication, utilizing a small checker core to monitor the computations performed by a larger microprocessor.

Rather than employing full hardware replication, recent work has also been interested in using smaller, lightweight hardware structures to target individual components of the processor. Argus [15] relies on a series of hardware checker units to perform online invariant checking to ensure correct application execution. In [22] Reddy and Rotenberg propose simple checkers that verify the functionality of decode and fetch units by comparing dynamically generated signatures, for small traces of identical instructions. They extend this idea in [23] by introducing additional checkers for other microarchitectural structures.

Although the area overhead of solutions like DIVA and Argus are significantly lower than full DMR, they still remain an expensive choice for commodity systems. Nevertheless, these nMR (and partial nMR) solutions provide greater fault coverage than Shoestring and can provide bounds on detection latency.

Redundant Multithreading (RMT): The introduction of simultaneous multithreading (SMT) capabilities in modern processors gave researchers another tool for redundant execution. Rotenberg's paper on AR-SMT [27] was the first to introduce the concept of RMT on SMT cores. The basic idea was to use the processor's extra SMT contexts to run two copies of the same thread, a leading thread and a trailing thread. The leading thread places its results in a buffer, and the trailing thread verifies these results and commits the executed instructions. Subsequent works improved upon this scheme by optimizing the amount of duplicated computation introduced by the redundant thread [7, 21, 24]. RMT has also been attempted at the software level by Cheng et al. [34]. This eliminates the need for architectural modifications to support RMT, and relies on the compiler to generate redundant threads that can run on general-purpose chip multiprocessors. With the advent of multicore, RMT solutions have evolved into using two or more discrete cores within a CMP to mimic nMR behavior. Reunion [31] and Mixed-mode reliability [40] are two recent proposals that allow idle cores within a CMP to be leveraged for redundant thread execution. The chief attraction of RMT approaches is the high cov-

Table 2: Shoestring compared to existing solutions for soft errors.

Solution	Hardware support	Software support	Perf. overhead	Area overhead	Coverage
n-Modular redundancy	YES	NO	LOW	VERY HIGH	VERY HIGH
Redundant Multi-threading	YES	MAYBE	HIGH	HIGH	VERY HIGH
Software instruction duplication	NO	YES	HIGH	NONE	HIGH
Symptom-based detection	NO	NO	LOW	NONE	MODERATE
Register file protection	YES	NO	LOW	MODERATE	MODERATE
Shoestring	NO	YES	LOW	NONE	HIGH

erage they can provide. The drawbacks of RMT include significant throughput degradation (loss of an SMT context or an entire core), hardware complexity/overhead, and potentially double the power consumption of non-RMT execution.

Software instruction duplication: Redundant execution can also be achieved in software without creating independent threads as shown by Reis et al. [25]. They proposed SWIFT, a fully compiler based software approach for fault tolerance. SWIFT exploits wide, underutilized processors by scheduling both original and duplicated instructions in the same execution thread. Validation code sequences are also inserted by the compiler to compare the results between the original instructions and their corresponding duplicates. CRAFT [26] and PROFIT [26] improve upon the SWIFT solution by leveraging additional hardware structures and architectural vulnerability factor (AVF) analysis [18], respectively. As in the case of RMT, compiler-based instruction duplication also delivers nearly complete fault coverage, with the added benefit of requiring little to no hardware cost. However, in order to achieve this degree of fault coverage, solutions like SWIFT can more than double the number of dynamic instructions for a program, incurring significant performance and power penalties.

Symptom-based detection: As mentioned in previous sections, Wang et al. was the first to exploit anomalous microarchitectural behavior to detect the presence of a fault. Their light-weight approach for detecting soft errors, ReStore [35, 37], leveraged symptoms including memory exceptions, branch mispredicts, and cache misses. The concept of anomaly detection has been further explored by Racunas et al. [20] who proposed verifying data value ranges and data bit invariants. Lastly, Li et al. [11] extended symptom-based fault coverage and applied it to detecting and diagnosing *permanent* hardware faults. The strength of symptom-based detection lies in its low-cost and ease of application. Unfortunately, the achievable fault coverage is limited and not appropriate for high error-rate scenarios.

Register file protection schemes: The register file holds a significant portion of program state. Consequently, error-free execution of a program cannot be accomplished without protecting it against faults. Just as main memory can be augmented with ECC, register file contents can also be protected by applying ECC. This process can be further optimized by protecting only live program variables, which usually occupy only a fraction of the register file. Solutions like the one presented by Montesinos et al. [16] builds upon this insight and only maintains ECC for those registers most likely to contain live values. Similarly Blome et al. [4] proposes a register value cache that holds duplicates of live register values. It is important to note that these schemes in general can only detect faults that occur after *valid* data has been written back to the register file. In contrast, Shoestring can also detect faults in other parts of the datapath that corrupt instruction output before it is written back to the register file or correction codes have been properly generated.

7. Conclusion

If technology scaling continues to exacerbate the challenges posed by transient faults, the research community cannot remain focused only on ultra-high reliability systems. We must devote efforts also to developing new innovative solutions for mainstream commodity processors. This paper introduces Shoestring, a transparent, software-based reliability solution that leverages both symptom-based detection as well as selective instruction duplication to minimize user-visible failures induced by soft errors. For a total performance penalty of 15.8%, Shoestring can cover an additional 33.9% of faults undetected by a conventional symptom-based scheme. Allowing just 1.6% of faults to manifest as user-visible data corruption, Shoestring is a cost-effective means of providing acceptable soft error resilience at a cost that the average commodity system can afford.

8. Acknowledgements

We thank the anonymous referees for their valuable comments and suggestions. We also owe thanks to Amir Hormati and Mark Woh for their feedback during initial discussions of this work as well as Emery Berger for his efforts as shepherd. This research was supported by National Science Foundation grants CCF-0916689 and CCF-0347411 and by ARM Limited.

References

[1] T. Austin. Diva: a reliable substrate for deep submicron microarchitecture design. In *Proc. of the 32nd Annual International Symposium on Microarchitecture*, pages 196–207, 1999.

[2] W. Bartlett and L. Spainhower. Commercial fault tolerance: A tale of two systems. *IEEE Transactions on Dependable and Secure Computing*, 1(1):87–96, 2004.

[3] D. Bernick, B. Bruckert, P. D. Vigna, D. Garcia, R. Jardine, J. Klecka, and J. Smullen. Nonstop advanced architecture. In *International Conference on Dependable Systems and Networks*, pages 12–21, June 2005.

[4] J. A. Blome, S. Gupta, S. Feng, S. Mahlke, and D. Bradley. Cost-efficient soft error protection for embedded microprocessors. In *Proc. of the 2006 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 421–431, 2006.

[5] S. Borkar. Designing reliable systems from unreliable components: The challenges of transistor variability and degradation. *IEEE Micro*, 25(6):10–16, 2005.

[6] F. A. Bower, D. J. Sorin, and S. Ozev. A mechanism for online diagnosis of hard faults in microprocessors. In *Proc. of the 38th Annual International Symposium on Microarchitecture*, pages 197–208, 2005.

[7] M. Gomaa and T. Vijaykumar. Opportunistic transient-fault detection. In *Proc. of the 32nd Annual International Symposium on Computer Architecture*, pages 172–183, June 2005.

[8] M. A. Gomaa, C. Scarbrough, I. Pomeranz, and T. N. Vijaykumar. Transient-fault recovery for chip multiprocessors. In *Proc. of the 30th Annual International Symposium on Computer Architecture*, pages 98–109, 2003.

[9] J. Kim, N. Hardavellas, K. Mai, B. Falsafi, and J. C. Hoe. Multi-bit Error Tolerant Caches Using Two-Dimensional Error Coding. In *Proc. of the 40th Annual International Symposium on Microarchitecture*, 2007.

[10] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proc. of the 2004 International Symposium on Code Generation and Optimization*, pages 75–86, 2004.

[11] M.-L. Li, P. Ramachandran, S. K. Sahoo, S. V. Adve, V. S. Adve, and Y. Zhou. Understanding the propagation of hard errors to software and implications for resilient system design. In *16th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 265–276, 2008.

[12] X. Li and D. Yeung. Application-level correctness and its impact on fault tolerance. In *Proc. of the 13th International Symposium on High-Performance Computer Architecture*, pages 181–192, Feb. 2007.

[13] T. May and M. Woods. Alpha-particle-induced soft errors in dynamic memories. *IEEE Transactions on Electron Devices*, 26(1):2–9, Jan. 1979.

[14] S. McCamant and M. D. Ernst. Quantitative information flow as network flow capacity. In *Proc. of the SIGPLAN '08 Conference on Programming Language Design and Implementation*, pages 193–205, June 2008.

[15] A. Meixner, M. Bauer, and D. Sorin. Argus: Low-cost, comprehensive error detection in simple cores. *IEEE Micro*, 28(1):52–59, 2008.

[16] P. Montesinos, W. Liu, and J. Torrellas. Using register lifetime predictions to protect register files against soft errors. In *Proc. of the 2007 International Conference on Dependable Systems and Networks*, pages 286–296, 2007.

[17] S. S. Mukherjee, M. Kontz, and S. K. Reinhardt. Detailed design and evaluation of redundant multithreading alternatives. In *Proc. of the 29th Annual International Symposium on Computer Architecture*, pages 99–110, 2002.

[18] S. S. Mukherjee, C. Weaver, J. Emer, S. Reinhardt, and T. Austin. A systematic methodology to compute the architectural vulnerability factors for a high performance microprocessor. In *International Symposium on Microarchitecture*, pages 29–42, Dec. 2003.

[19] N. Oh, S. Mitra, and E. J. McCluskey. Ed4i: Error detection by diverse data and duplicated instructions. *IEEE Transactions on Computers*, 51(2):180–199, 2002.

[20] P. Racunas, K. Constantinides, S. Manne, and S. Mukherjee. Perturbation-based fault screening. In *Proc. of the 13th International Symposium on High-Performance Computer Architecture*, pages 169–180, Feb. 2007.

[21] V. Reddy, S. Parthasarathy, and E. Rotenberg. Understanding prediction-based partial redundant threading for low-overhead, high-coverage fault tolerance. In *14th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 83–94, Oct. 2006.

[22] V. Reddy and E. Rotenberg. Inherent time redundancy (itr): Using program repetition for low-overhead fault tolerance. In *Proc. of the 2007 International Conference on Dependable Systems and Networks*, pages 307–316, June 2007.

[23] V. Reddy and E. Rotenberg. Coverage of a microarchitecture-level fault check regimen in a superscalar processor. In *Proc. of the 2008 International Conference on Dependable Systems and Networks*, pages 1–10, June 2008.

[24] S. K. Reinhardt and S. S. Mukherjee. Transient fault detection via simulataneous multithreading. In *Proc. of the 27th Annual International Symposium on Computer Architecture*, pages 25–36, June 2000.

[25] G. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August. SWIFT: Software implemented fault tolerance. In *Proc. of the 2005 International Symposium on Code Generation and Optimization*, pages 243–254, 2005.

[26] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, D. I. August, and S. S. Mukherjee. Software-controlled fault tolerance. *ACM Transactions on Architecture and Code Optimization*, 2(4):366–396, 2005.

[27] E. Rotenberg. AR-SMT: A microarchitectural approach to fault tolerance in microprocessors. In *International Symposium on Fault Tolerant Computing*, pages 84–91, 1999.

[28] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *Tenth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 45–57, New York, NY, USA, 2002. ACM.

[29] P. Shivakumar, M. Kistler, S. Keckler, D. Burger, and L. Alvisi. Modeling the effect of technology trends on the soft error rate of combinational logic. In *Proc. of the 2002 International Conference on Dependable Systems and Networks*, pages 389–398, June 2002.

[30] J. Smolens, J. Kim, J. Hoe, and B. Falsafi. Efficient resource sharing in concurrent error detecting superscalar microarchitectures. In *Proc. of the 37th Annual International Symposium on Microarchitecture*, pages 256–268, Dec. 2004.

[31] J. C. Smolens, B. T. Gold, B. Falsafi, and J. C. Hoe. Reunion: Complexity-effective multicore redundancy. In *Proc. of the 39th Annual International Symposium on Microarchitecture*, pages 223–234, 2006.

[32] L. Spainhower and T. Gregg. IBM S/390 Parallel Enterprise Server G5 Fault Tolerance: A Historical Perspective. *IBM Journal of Research and Development*, 43(6):863–873, 1999.

[33] N. Vachharajani, M. J. Bridges, J. Chang, R. Rangan, G. Ottoni, J. A. Blome, G. A. Rei, M. Vachharajani, and D. I. August. Rifle: An architectural framework for user-centric information-flow security. In *Proc. of the 37th Annual International Symposium on Microarchitecture*, pages 243–254, Dec. 2004.

[34] C. Wang, H. seop Kim, Y. Wu, and V. Ying. Compiler-managed software-based redundant multi-threading for transient fault detection. In *Proc. of the 2007 International Symposium on Code Generation and Optimization*, 2007.

[35] N. Wang and S. Patel. Restore: Symptom based soft error detection in microprocessors. In *International Conference on Dependable Systems and Networks*, pages 30–39, June 2005.

[36] N. J. Wang, M. Fertig, and S. J. Patel. Y-branches: When you come to a fork in the road, take it. In *Proc. of the 12th International Conference on Parallel Architectures and Compilation Techniques*, pages 56–65, 2003.

[37] N. J. Wang and S. J. Patel. ReStore: Symptom-based soft error detection in microprocessors. *IEEE Transactions on Dependable and Secure Computing*, 3(3):188–201, June 2006.

[38] N. J. Wang, J. Quek, T. M. Rafacz, and S. J. Patel. Characterizing the Effects of Transient Faults on a High-Performance Processor Pipeline. In *International Conference on Dependable Systems and Networks*, page 61, June 2004.

[39] C. Weaver and T. M. Austin. A fault tolerant approach to microprocessor design. In *Proc. of the 2001 International Conference on Dependable Systems and Networks*, pages 411–420, Washington, DC, USA, 2001. IEEE Computer Society.

[40] P. M. Wells, K. Chakraborty, and G. S. Sohi. Mixed-mode multicore reliability. In *17th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 169–180, 2009.

[41] M. T. Yourst. Ptlsim: A cycle accurate full system x86-64 microarchitectural simulator. In *Proc. of the 2007 IEEE Symposium on Performance Analysis of Systems and Software*, pages 23–34, 2007.

[42] J. F. Ziegler and H. Puchner. *SER-History, Trends, and Challenges: A Guide for Designing with Memory ICs*. Cypress Semiconductor Corp., 2004.