Reducing the Cost of Protection against Soft Errors using Profile-Based Analysis

Daya S. Khudia
The University of Michigan
dskhudia@umich.edu

Griffin Wright
The University of Michigan
grwright@umich.edu

Scott Mahlke
The University of Michigan
mahlke@umich.edu

Abstract

The negative impact of the aggressive scaling of technology nodes on the sensitivity of CMOS devices to soft errors has been well studied in the past. Technology scaling makes processors more susceptible to transient faults. Errors caused by high-energy particle strikes in processors can result in unexpected behavior and incorrect results. With the smaller and cheaper CMOS devices pervasive in mainstream computing, it is necessary to protect these devices against soft errors; an increasing rate of faults makes the need to protect the critical application running on commodity processors against soft errors more important than ever. The existing methods of protecting against such faults generally have high area or performance overheads and so are used only in mission-critical domains. In order to protect against the soft errors caused by high energy particle strikes, the detection of these errors is necessary so that recovery can be triggered.

In this work, a profile-based software-only application analysis and transformation solution to detect soft-errors is presented. The goal is to develop a low cost solution which can be deployed for off-the-shelf commodity processors. The solution works by intelligently duplicating instructions which are likely to affect the program output, and comparing results between original and duplicated instructions. The intelligence of our solution is provided by the use of control flow, memory dependence, and value profiling to understand and exploit the common-case behavior of applications. Our solution is able to achieve 92% fault coverage with a 20% instruction overhead. This represents a 41% lower performance overhead than the best prior approaches with approximately the same fault coverage.

General Terms Transient Fault Reliability, Soft Errors, Profile-based Compiler Analysis, Fault Injection

1. Introduction

Any computer system is expected to work reliably during its lifetime. Modern computer systems are built using billions of tiny transistors, and even a single transistor failure can render a computer system useless. Most hardware vendors have a lifetime reliability target to achieve an acceptable product quality, and we focus on that target.

The focus of this work is soft errors, or single-event-upsets (SEUs). Soft errors, also referred to as transient faults, are caused by neutrons from cosmic radiation and alpha particles from packaging material impurities. As the name suggests, transient faults are not persistent and do not render the computer system unusable for its lifetime. However, when a transient fault occurs in a computer system, it has the ability to corrupt the application output or crash the system.

Commodity computer systems have relatively tight cost budgets because of intense competition. In these markets, 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, reliability requirements are also not stringent; Consumers can tolerate glitches in video playback and infrequent crashes of their desktop/laptop computers (usually caused by software bugs). The key challenge facing the consumer electronics market in future deep sub-micron technologies is providing just enough coverage of soft errors so that the effective fault rate (the raw SER scaled by the available coverage) remains at levels to which people have become accustomed. Providing solutions which can achieve this coverage "on the cheap" is the goal of this work.

To achieve statistically significant soft error coverage at minimal overheads, we propose a software-centric approach for detecting soft errors. This work 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 [9, 16]. These symptoms can include memory access exceptions, mispredicted branches, and even cache misses. Although symptom-based detection is inexpensive, the amount of coverage that can be obtained from a symptom-only approach is typically limited. To address this limitation, we make use of the second area of prior research, software-based instruction duplication [13, 14]. 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 more than 90% 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 this work 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 masked by light-weight symptom-based detection. To address the remaining faults, compiler analysis is applied 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. Our solution intelligently selects between relying on symptoms and judiciously applying instruction duplication to optimize the coverage and performance trade-off. In this way, our solution provides a low-cost, high-coverage solution for soft errors in processors targeted for the consumer electronics market. The contributions of this work are as follows:

- A software solution which does not need any user annotations in an application to generate reliability-aware code and which works on applications written in variety of languages.
- A new reliability-aware compiler analysis that uses various forms of profiling (memory profiling, value profiling etc.) to identify instructions which would not be covered by symptom-based fault detection alone.
- A selective instruction duplication approach that leverages memory profiling and value profiling in compiler analysis to identify and replicate a small subset of vulnerable instructions.
- Microarchitectural fault injection experiments to demonstrate the effectiveness of our proposed solution in terms of fault coverage and performance overhead.

2. Background and Motivation

Feng et al. [4] and Shivakumar et al. [15] presented data for the effect of technology scaling on the *failures in time* (FIT[1]) metric. They showed an exponential increase in the SER for future technology generations. Since for future technologies it will be hard to power on all the transistors at once, aggressive voltage scaling is expected to be used. Voltage scaling further exacerbates the problem of soft errors as smaller disturbances in circuits will be able to flip a bit.

2.1 Instruction Duplication and Terminology

SWIFT [13] proposed the idea of duplicating instruction in a single thread of execution. SWIFT recursively duplicated instructions by walking the data flow chains of stores and by protecting control flow. Shoestring [4] improved upon this idea by considering only global stores and by protecting the control flow only for the immediate branch that affects the execution of a global store instruction. For classifying instructions, we adopt the terminology used by Shoestring. The initial analysis phase of our solutions classifies instructions into the categories described below.

- **Symptom-generating**: these instructions, if they consume a corrupted input, are likely to produce detectable symptoms.
- **High-value**: instructions which are likely to corrupt the output of the program if they consume a corrupted input.
- **Safe**: these instructions are naturally covered by symptom-generating consumers.

In our solution, all library call and function call are considered high-value instructions. Starting from the operands of these high-value instructions, we recursively duplicate producers of the operands. This recursive duplication is terminated when 1) No more producers exist, 2) A safe instruction is encountered, or 3) the producers are already duplicated. The safe instructions are determined based on the probability of whether or not a particular instruction would generate a symptom if corrupted.

2.2 Proposed Solution Landscape

A soft error solution that targets the commodity user space needs to be designed with lower overhead and acceptable coverage as targets. Two commonly employed techniques are symptom-based and duplication-based fault detection schemes. Our solution is a hybrid of these two techniques and tries to achieves as much fault coverage as possible by leveraging the strengths of each technique.

Symptom-based detection approaches rely mostly on hardware exceptions, and their coverage quickly saturates. The saturation of fault coverage provided by symptom-based methods is expected because these schemes rely on rare hardware exceptions such as page fault, divide-by-zero, etc. If more frequently occurring microarchitectural events such as branch mispredicts and cache misses are included as symptoms for starting recovery, then recovery can be triggered more frequently and the overhead can become unacceptable [16]. Symptom-based methods provide good coverage with less overhead.

The coverage versus performance curve is far less steep for instruction duplication; Coverage increases almost linearly with the amount of code duplication. One advantage of instruction-based duplication is that the amount of coverage can be tuned according to an application's requirements by providing more or less duplication of code.

Neither symptom-based nor instruction duplication-based techniques provide a stand-alone solution to achieve the desired coverage and performance benefits. The proposed solution in this work tries to strike a balance between performance overhead and fault coverage by exploiting the strengths of each technique.

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

2.3 Opportunities for Profile-Based Duplication

In the past, profiling information has been successfully used in profile-guided optimizations (PGOs) to improve the performance of a program [5]. GCC [6] and Intel's compiler (icc) can use profiling information to generate an efficient program binary. Most optimizations based on profiling data work by uncovering previously unexplored opportunities. For example, if a multiply operation generates the same invariant value frequently, then the multiply operation can be optimized away with a check inserted for the correct value. Similarly, edge profiling and memory profiling can be used in optimizations such as partial dead-code-elimination, improved object layout, and more.

In this paper we use edge-profiling, memory profiling and value profiling for the first time (to the best of our knowledge) in the context of code duplication for protection against soft errors. With the profiling information we can exploit the common case behavior of a program to duplicate only the critical instructions, which need to be protected. Different kinds of profiling information enable us to ignore unnecessary duplication of instructions which are unlikely to cause program output corruption in the presence of a transient fault. For example, in the context of having the same invariant value generated by an instruction, we insert a comparison with the specific invariant value in the code. The failure of this comparison then indicates the possibility of a transient fault and triggers the recovery mechanism via a jump to recovery code. The following describes the scenarios where profiling information can be used.

- **Through-Memory Analysis**: While recursively duplicating instructions starting from high-value instructions, the duplication will terminate when one of three termination conditions (Section 2.1) are met. However, when a load is encountered in this duplication process, we can trace the dependences that manifest through memory by using memory profiling. Our work implements through-memory analysis to ensure we maximize the efficiency of our duplicated code and maintain a superior level of coverage at a minimal overhead.
- **Value Profiling**: Previous symptom-based works have explored unusual scenarios such as erroneous hardware behavior to detect faults. We take this to the next level by detecting more abnormal scenarios with value profiling. We also eliminate the unnecessary duplication of instructions which provide a low rate of return with regards to additional fault coverage by using control flow and memory profiling.

3. Proposed Solution

The main underlying observation behind our proposed solution is that 100% reliability is not always required. For example, when a user is playing a game or watching a video, a few infrequent soft errors are tolerable as they do not lead to user-apparent anomalies. Our proposed solution leverages the basic idea of instruction duplication from SWIFT and others [12, 13], and adds intelligent tracing of dependences through memory and other profiling information to generate more efficient duplication code. In essence, our solution uses the dynamic behavior of applications to generate efficient code for transient fault detection. The remaining sections describe our newly proposed techniques and the duplication process.

3.1 Overview of proposed solution

The main intuition behind our idea is that applications mainly communicate to the external world using I/O calls, and if we can capture the true dependences of the operands of these calls, we can better protect the program output from getting corrupted. This type of approach is suitable for our low overhead approach as we don't target 100% fault coverage.

To capture the true dependences of the operands of library calls, we use LAMP [10], which allows us to determine the dependences that manifest through memory. In contrast to Shoestring [4], while duplicating instructions, our solution traces the dependences through memory. In the recursive duplication of the producer chains of the operands of high value instructions, whenever a load is encountered, Shoestring terminates this recursive duplication at that load, and does not consider the dependences that manifest through memory. Our solution deviates from Shoestring in this duplication process to achieve better and more useful code duplication. In our solution, the duplication process starts from the operands of library calls. If a load is encountered during duplication, the compiler pass obtains all the stores that wrote to the address from which the load is reading using the memory profiling information. The duplication process considers these stores as potential candidates for instructions can corrupt program output. The producer chains of these stores are also protected by duplication.

3.2 Overhead Reduction Without Losing Coverage

Our solution detects soft errors by adding extra instructions in the current thread of execution, incurring a penalty in performance. Using various kinds of profiling information, we can reduce this overhead. In particular, we utilize edge profiling for not protecting infrequently executed instructions, memory profiling to find load and store aliases and identify silent stores, and value profiling to get the information about instructions which produce statistically invariant values. The performance overhead incurred because of instruction duplication can be reduced further by using information about the runtime behavior of applications through profiling. Information about the runtime behavior of programs enables us to remove duplication for protecting the code that doesn't provide significant fault coverage.

3.2.1 Execution Frequency Based Recursive Duplication Termination

The intuition behind this optimization is that frequently executed instructions should not be duplicated to protect a non-frequently executed instruction. The probability of a soft error affecting a non-frequently executed instruction is relatively low and so to protect such a instruction, unnecessary duplication of frequently executed instructions should not be performed.

3.2.2 Dependences Through Memory

We use memory profiling to obtain information about aliasing between loads and stores. As pointed out in Section 3.1, to duplicate the true dependences of the producer chains of high value instructions, we need to have load/store dependence information available. Memory profiling provides us with this information. If we have the memory profiling information available at the time of duplication, intelligent duplication can be performed. For example, only library calls can be considered as high value instructions and only the stores that alias with the loads in the producer chain of library call operands need to be protected.

3.2.3 Silent Store Optimization

Memory profiling is also used to identify silent stores that exist in an application. A silent store is defined as a store which writes the same value to a memory location that is already present at that location. As reported in many previous studies, a significant percentage of total stores are silent. Bell et al. [1] report 18% to 64% of total stores as silent for SPEC95 benchmarks. We have implemented silent store profiling as an extension of the LAMP tool [10]. In experiments with SPECINT2000 benchmarks, we observed silent stores ranging from 0.01% to 72% of total stores. The presence of high fractions of silent stores can be exploited.

For the purpose of this work, while doing recursive duplication, if we encounter a store which is silent then we stop the recursive duplication. Considering the high percentage of stores that exist in benchmark applications, we can save in terms of instruction duplication. The intuition behind this idea is that even if a corrupted value is written by a store it will be written correctly in subsequent executions of the same store. If a load is present between two dynamic instances of a silent store; a fault in the store can cause the load to read the wrong value. So, whenever an aliasing pair load and a silent store are part of the same loop, the store is not considered for this optimization.

3.2.4 Software Symptom Generation using Value Profiling

If an instruction generates the same value almost 100% of the time, we can use that value to compare to the value generated by the same instruction at runtime. If the value generated at runtime differs from the one that the instruction generates very frequently, it is assumed that a fault has occurred and the recovery mechanism is triggered. Since for each value comparison we need to insert one compare (cmp) and one branch instruction, these instructions should be only inserted when they provide benefits in comparison to straight up duplication of the data flow chain. The benefits are only present if the data flow chain is long and the count of instructions which would have been duplicated is greater than 2 (the value comparison + the branch instruction).

If the two compared values do not match at runtime, then the recovery mechanism is triggered. Although rare, it is possible that at runtime, the application encounters different inputs and so an instruction may produce output other than the profiled value. Since this is a rare case, the recovery should be initiated only once from the same place; if the comparison fails at a location twice from the same place, such requests for recovery are ignored.

4. Experimental Setup

The experiments with high energy particle strikes conducted by Dixit et al. [3] are not feasible in academic studies such as the one presented in this paper. An acceptable alternative to these experiments is the use of statistical fault injections (SFI) into a microarchitectural model of a processor. SFI has been previously used in validating the solutions proposed to solve the problem of soft errors. For the purpose of this work, we use a single bit-flip fault model implemented in the microarchitectural model of a ARM processor.

For profiling the SPECINT2000 benchmarks we have used training data provided in the benchmark suite corresponding to each benchmark. While running the benchmark on the simulator, we utilized ref data provided in the benchmark suite. We only use training data for profiling. However, profiling information from multiple runs of a program with representative inputs can be combined easily in our profiling infrastructure.

4.1 Compiler Passes

We have used the LLVM [7] compiler infrastructure to implement the reliability-aware code generation pass. This pass uses internal information from other analysis passes such as memory profiling and value profiling to produce bitcode with duplicated instructions. The LLVM code generation framework is then used to generate ARM binaries from the bitcode with duplicated instructions. Few other optimization passes, especially in the code generation phase, can remove the duplicated instructions, but we did disable the machine common subexpression elimination pass and a few others that are run while preparing the IR (Intermediate Representation) for code generation.

Since LLVM supports a number of front-ends (including C/C++), the developed pass is capable of generating reliability-aware code for applications written in many languages. The pass takes LLVM IR as input and also produces IR with duplicated instructions. The other benefit of operating at the IR level is that all the code generation targets supported by LLVM (Alpha, ARM, etc.) can be used with the solution presented in this paper. We have performed all experiments targeting an ARM architecture. To instrument an application with reliability aware

code for a different target, our framework can remain the same and the machine executable can be generated for a multitude of targets.

4.2 Fault Injection Framework

The fault model used in this work is a single bit-flip model widely used in experimental evaluation of the previously proposed solutions to tackle the problem of soft errors. These faults are inserted by flipping a bit at a random cycle during the course of application run. The experimental results shown in this paper are produced with fault injection trials. At the start of each trial a random physical register and a random bit are selected for injection. The selected bit is then flipped at a random time during the application run and the program executes with this modified register data. For the purposes of this work, we have used the GEM5 [2] simulator. The simulator is running in ARM syscall emulation mode and models the ARMv7-a profile of ARM architecture. We have used a model of the in-order ARM architecture.

4.2.1 Register File Injections

To calculate the statistical significance of a given number of fault injection trials, we use the works of Leveugle et al. [8]. We need 96 fault injection trials for each benchmark to have a 10% margin of error and confidence level of 95%. Therefore, we chose 100 fault injection trials for each benchmark to yield results with reasonable accuracy in a timely manner. For the experiments, we injected faults randomly into the register file. It has been pointed out in a previous study [17] that many corruptions are due to injections into the register file, making it an attractive target for injection studies. Furthermore, Wang et al. [17] showed that the bulk of transient fault-induced failures are dominated by corruptions introduced from injections into the register file.

4.3 Outcome Classification

After the fault injection, the program runs until completion and the log files are collected. So, the experiments represent full runs of the benchmarks. At the end of every simulation, the log files are analyzed to determine the outcome of the run as described below. The result of each trial is classified into one of four categories:

1. **Masked**: The injected fault did not corrupt the program output. Application-level or architecture level masking occurred in this case.
2. **Covered by symptoms**: The injected fault produces a symptom such as a page fault or divide-by-zero fault within 1000 cycles of the fault injection so that a recovery can be triggered. Section 4.4 describes the recovery support in further detail.
3. **SWDetect**: The injected fault was detected by the extra comparison inserted at the time of duplication.
4. **Failures**: All faults that cause early program termination or do not terminate in definite time are classified into this category.
5. **SDCs**: These are silent data corruptions (SDCs). All faults that produce user-visible corruptions are classified into this category.

The result classifications of the injection experiments in this work are based on the fact that only user-visible corruptions really matter. The cost of ensuring reliability can be reduced by focusing on hiding only the faults that are noticeable by the end user at run-time. Therefore, the metric of importance is not the number of faults that propagate into the microarchitectural state, but rather the percentage of faults that actually do result in user-visible failures.

4.4 Recovery Support

Our solution 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 1000 instructions in the past will already be required for aggressive performance speculation. Consistent with Wang and Patel [16], our work assumes that any fault that manifests as a symptom within a window of 1000 committed instructions can be safely detected and recovered.

4.5 Benchmarks

We have used all except 2 applications from the SPEC2K integer benchmark suite (*gzip, vpr, gcc, mcf, crafty, perlbmk, parser, gap, vortex, bzip2*) as representative workloads in experiments, and they are compiled with standard -O3 optimizations. The remaining two benchmarks (*eon* and *twolf*) did not work with the cross-build of LLVM to produce ARM binaries on an x86 machine. In this work, multi-threaded programs are not considered. However, we do not foresee any problems of using our technique with race-free multi-threaded programs. Code duplication in a multi-threaded environment may uncover hidden concurrency bugs because the extra duplicated instructions inserted may change the relative ordering of instructions in the simultaneous execution of threads.

5. Experimental Results

This section presents the combined results of our technique, which combines the use of silent store information (to eliminate unnecessary duplication), edge profiling, value profiling, and memory profiling (to trace dependences through memory) techniques.

5.1 Silent Stores

The use of silent store information yielded an average of roughly 12% overhead reduction across all benchmarks with an average dynamic silent store percentage of 31%, though 176.gcc, 181.mcf, 253.perlbmk and 255.vortex had a higher percentage of dynamic silent stores and yielded a more significant reduction in overhead. Duplication is terminated (see Section 3.2.3) only when a static store is silent more than 80% of the time (i.e., if a static store in a benchmark writes the same value already present at a memory location less than 80% of its dynamic execution time, the store is not considered for this optimization). 175.vpr and 253.perlbmk have less of a reduction in overhead because many static stores in these benchmarks do not cross the threshold of 80%.

5.2 Performance overheads and Fault Coverage

In this subsection, a comparison of our solution is made with Shoestring (profile-oblivious duplication) using the criteria of performance overhead and fault coverage.

5.2.1 Effect of safe instructions and control flow protection

We examined the maximum amount of coverage we could obtain by doing the maximum amount of duplication. Since loads are never duplicated to save on memory traffic, there will always be some faults which can escape detection by the duplicated code, including those due to faults injected into duplicated instructions. The full duplication column in Figure 1 shows the performance overhead and fault coverage if the duplication is not terminated at safe instructions and all the branches are also protected by duplication. The full-dup column in Figure 2 is the corresponding fault coverage breakdown among the different categories of result classification. Essentially, Full duplication data represents the performance overhead and fault coverage with the maximum amount of duplication possible with our scheme. On average, the performance overhead is 50.51% and the coverage of transient faults by combining symptom-based and duplication-based methods is 92.2%. Though the overhead is high, it gives improved coverage of faults. In the 164.gzip benchmark, all unmasked faults are detected by the duplicated code.

The profile-oblivious duplication column in Figure 1 and pro-oblivi column in Figure 2 show the performance overhead and fault coverage numbers if the duplication is terminated at safe instructions and

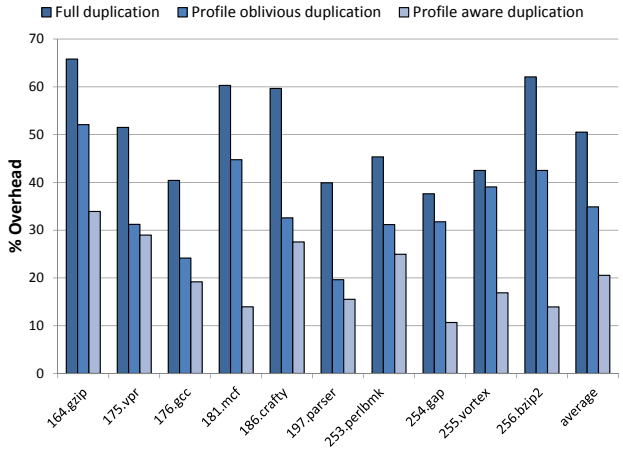

Figure 1: Overhead comparison among full duplication, profile oblivious duplication, and profile-aware duplication. In full duplication, duplication is not terminated at safe instructions and all branches are also protected. Although profile oblivious duplication uses safe instructions, profiling information is not utilized. Profile-aware duplication uses safe instructions as well as profiling information.

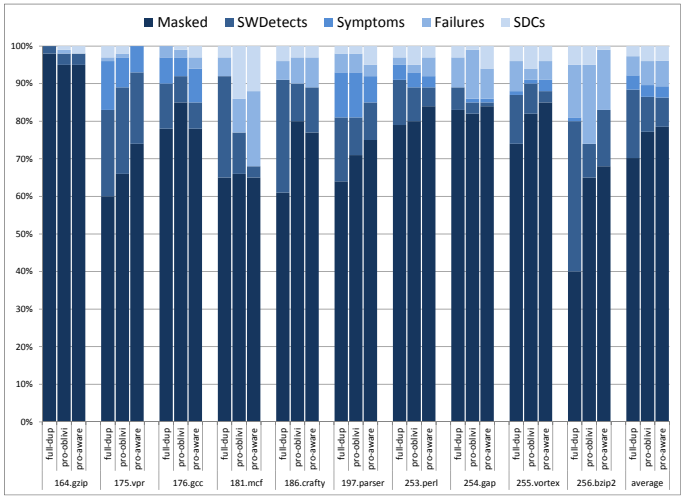

Figure 2: Coverage breakdown for full duplication (full-dup), profile oblivious duplication (pro-oblivi) and profile aware duplication (pro-aware).

only the immediate branch whose execution affects the execution of high value instruction is protected by duplication. These two techniques (safe instructions and immediate branch protection), also implemented in Shoestring, reduce overhead but also decrease fault coverage from 92.2% to 89.6%. These are Shoestring performance and coverage numbers, and we have considered these as our baseline values for result comparisons in the remainder of this section.

5.2.2 Through-memory dependencies and profiling usage

The profile-aware duplication column in Figure 1 shows the overhead if we duplicate the producer chains of library calls only (i.e., only library calls are considered as high value instructions) and make use of profiling information. The pro-aware column in Figure 2 shows the corresponding coverage breakdown numbers. In this set of experiments, the effectiveness of using LAMP to trace the dependences through memory and other profiling techniques while duplicating instructions is demonstrated. The overhead is reduced by 41% but the coverage of transient faults provided by the combination of symptom-based and software duplication stays about the same. These results demonstrate the effectiveness of using the profiling information for efficient duplication. This technique results in better code duplication, providing the same level of fault coverage seen at 34.9% overhead with Shoestring but with a 41% lower overhead.

5.3 Contributions of Each Technique

So far we have discussed the combined effect of edge, memory, and value profiling on the obtained results. In this section, the contribution of each technique is briefly mentioned; space constraints prevent us from showing individual graphs for results.

Our data show that if the silent store and recursive duplication termination optimizations are used, there is a 12.78% reduction in overhead with a combined fault coverage similar to that provided by Shoestring. When applying the silent store optimization (Section 3.2.3) and the recursive duplication break optimization (Section 3.2.1), the average coverage provided is reduced by 3%, but the overhead is reduced by 13%. Thus, these optimizations improve the end result of the overall technique. Finally, the use of value profiling was found to provide an average of 5.9% reduction in the performance overhead of duplication, as well as providing a slight increase in the number of faults covered by duplication.

Overall, the experimental results demonstrate that the techniques proposed in this work are effective as they provide a significant reduction in performance overhead while still maintaining the desired fault coverage levels. Performance overheads shown in this paper for Shoestring are higher than originally reported in the Shoestring paper because our infrastructure is different than the one used in evaluating Shoestring. For example, we target an ARM in-order architecture while Shoestring used out-of-order AMD architecture.

5.4 Cases of SDCs

In our work, loads are never duplicated to save on memory traffic. Consider the case where an injected fault corrupts the data in a register loaded by a load. Since the duplicated data flow chain and the original data flow chain both will use the same corrupted data, this fault will not be detected. This is the most frequent cause of the SDCs observed.

6. Future Work

As part of ongoing and future work, we are exploring the following different aspects related to this work.

6.1 Branch Target Corruptions

We are investigating the effects of bit-flip corruptions in taken branch targets. Our data show that our current technique does not offer as much coverage when dealing with corrupted branch targets. Specifically, early-termination and fatal faults are much more common due to jumps to incorrect branch targets. We are currently working on enhanced signature tracking techniques to ensure valid and correct program flow.

6.2 Out of Order Framework

The current results are reported with an in-order model. Since our injection site is the register file, we expect that an out-of-order model would not affect our conclusions significantly. In fact, an out-of-order model actually improves our coverage rates because duplication of instructions in a single thread of execution results in extra instruction level parallelism which an out-of-order model could exploit efficiently. In addition, the inherent ability of an out-of-order model to squash some erroneous instructions or roll-back after a branch mispredict allows for some further inherent masking abilities. Fault injection experiments with an out-of-order implementation are planned as part of our future work.

7. Related Work

Shoestring [4] is the closely related work to our proposed solution. Our work differs from Shoestring in the following ways:

- Our work makes use of value profiling to generate extra software based symptoms.
- Silent store profiling information is incorporated in this work for the first time.
- In Shoestring, all of the global stores and all functions calls are considered as high value. Our solution starts duplicating instructions only from library calls and then uses memory profiling to find the true load/stores dependencies. Shoestring terminated DFG traversing if a load was encountered while duplicating instructions. In this process, only the important stores get considered as high value and a lesser duplication overhead is achieved.
- Shoestring targets an x86 architecture and is specifically designed in the compiler back-end to operate at the level of machine functions and machine basic blocks. Our work takes a fresh approach, and is implemented instead at the IR (Intermediate Representation) level. This enables greater applicability, as IR-level implementation allows for a wider target base, being useable on a multitude of different processor architectures.

With respect to other hardware and software based solutions, our solution's ability to achieve improved levels of fault coverage with very low performance overhead, and all without any specialized hardware, sets it apart.

Software instruction duplication is an approach which is extended in our work in an effort to increase fault-coverage while reducing overhead and eliminating the need for additional hardware support. In this case, redundant execution can also be achieved in software without creating independent threads as shown by Reis et al. [13]. The authors 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. Other works such as CRAFT [14] and PROFIT [14] improve upon the SWIFT solution by leveraging additional hardware structures and architectural vulnerability factor (AVF) analysis, respectively.

Our proposed solution also makes use of symptom-based detection, which relies on anomalous microarchitectural behavior to detect soft errors. A light-weight approach for detecting soft errors, ReStore [16], analyzes symptoms including memory exceptions, branch mispredicts, and cache misses. In our proposed solution, extra symptom generating instructions are introduced based on value profiling data.

Yu et al. [18] explored the idea of identifying critical variables in program by repeatedly injecting faults into a program and then analyzing the cases that cause SDCs. Since for larger programs there can be many variables which affect program output and identifying them by injecting faults would require a large number of experiments, this approach is not scalable. Nakka et al. [11] presented a hardware-based solution for selective replication. Similar to high-value instructions, they identify critical variables. Their approach duplicates a number of hardware structures to provide instruction replication, while our software-only approach targets off-the-shelf commodity processors.

8. Conclusions

The ever-present desire to scale transistor size will increase the rate at which soft errors occur during the time when the processor is in use. As a result, it is necessary to provide protection against soft errors not only for mission-critical applications but also for important applications running on commodity processors. The high overhead of techniques to protect against soft errors for mission-critical computing systems is not acceptable for applications running on commodity processors. In this work, we presented a solution for commodity processors that uses profile-based compiler analysis to selectively duplicate instructions. Our profile-based selective duplication results in a reduction in overhead of 41% in comparison to a previously proposed solution while maintaining the same level of fault coverage.

References

[1] G. B. Bell, K. M. Lepak, and M. H. Lipasti. Characterization of silent stores. In *Proc. of the 9th International Conference on Parallel Architectures and Compilation Techniques*, 2000.

[2] N. L. Binkert, E. G. Hallnor, and S. K. Reinhardt. Network-oriented full-system simulation using M5. In *6th Workshop on Computer Architecture Evaluation using Commercial Workloads*, pages 36–43, Feb. 2003.

[3] A. Dixit and A. Wood. The impact of new technology on soft error rates. In *Reliability Physics Symposium (IRPS), 2011 IEEE International*, april 2011.

[4] S. Feng, S. Gupta, A. Ansari, and S. Mahlke. Shoestring: Probabilistic soft-error reliability on the cheap. In *18th International Conference on Architectural Support for Programming Languages and Operating Systems*, Mar. 2010.

[5] R. Gupta, E. Mehofer, and Y. Zhang. Profile guided compiler optimizations. *The Compiler Design Handbook: Optimizations and Machine Code Generation, CRC Press*, 2002.

[6] J. Hubicka. Profile driven optimisations in gcc. *GCC Summit Proceedings*, pages 107–124, 2005.

[7] 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.

[8] R. Leveugle, A. Calvez, P. Maistri, and P. Vanhauwaert. Statistical fault injection: quantified error and confidence. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '09, pages 502–506. European Design and Automation Association, 2009.

[9] 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.

[10] T. Mason. LAMPVIEW: A Loop-Aware Toolset for Facilitating Parallelization. Master's thesis, Dept. of Electrical Engineeringi, Princeton University, Aug. 2009.

[11] N. Nakka, K. Pattabiraman, and R. Iyer. Processor-level selective replication. In *Proc. of the 2007 International Conference on Dependable Systems and Networks*, 2007.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] J. Yu and M. J. Garzarn. A detector for harmful errors. *IEEE Workshop on Silicon Errors in Logic - System Effects*, 2009.