Efficient Soft Error Protection for Commodity Embedded Microprocessors using Profile Information

Daya Shanker Khudia, Griffin Wright, and Scott Mahlke

Advanced Computer Architecture Laboratory
The University of Michigan - Ann Arbor, MI
{dskhudia, grwright, mahlke}@umich.edu

Abstract

Successive generations of processors use smaller transistors in the quest to make more powerful computing systems. It has been previously studied that smaller transistors make processors more susceptible to soft errors (transient faults caused by high energy particle strikes). Such errors can result in unexpected behavior and incorrect results. With smaller and cheaper transistors becoming pervasive in mainstream computing, it is necessary to protect these devices against soft errors; an increasing rate of faults necessitates the protection of applications running on commodity processors against soft errors. The existing methods of protecting against such faults generally have high area or performance overheads and thus are not directly applicable in the embedded design space. In order to protect against soft errors, the detection of these errors is a necessary first step so that a recovery can be triggered.

To solve the problem of detecting soft errors cheaply, we propose a profiling-based software-only application analysis and transformation solution. The goal is to develop a low cost solution which can be deployed for off-the-shelf embedded processors. The solution works by intelligently duplicating instructions that are likely to affect the program output, and comparing results between original and duplicated instructions. The intelligence of our solution is garnered through 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.

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

General Terms Reliability, Soft Errors, Profiling

Keywords Profile-based Compiler Analysis, Fault Injection

1. Introduction

Any microprocessor-based computing system is expected to work reliably during its lifetime. A typical set of tasks performed on a commodity level computer system could include video games, web browsing, bank transactions, and more. While running these applications on their computers, users want their experience to be fault-free. 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.

The focus of this work is soft errors, or single-event-upsets (SEUs). Soft errors, also referred to as transient faults, are primarily caused by neutron particle strikes 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 can corrupt the application output or crash the system.

Soft errors due to packaging contamination have been reported for several decades. In 1978, Intel Corporation reported that chip packaging modules were contaminated with Uranium from a mine nearby [19]. Neutrons form the atmosphere were to blame in another incident in 1996, when E. Normand [24] detailed single event upsets in RAM chips. A third example of such errors was noted in 2004 by Cypress Semiconductor who claimed a number of incidents related to soft errors [36]. One single error resulted in the crash of a data center while another series of errors caused frequent shutdowns in a massive automotive factory.

The amount of charge released by high energy particle strikes determines whether a transistor will malfunction or not. If the size and operating voltage of transistors in a system is small, it is more likely to be affected by particle strikes. Transistor sizes and operating voltages are decreasing, making future technology generations more susceptible to soft errors [29]. Traditionally, reliability research has focused largely on the high-performance server market. Notable past works in this area have been the IBM S/360 (now Z-series servers) [2, 31] and the HP NonStop systems [4]. Both utilize large-scale modular redundancy for effective fault tolerance. As such, they are not feasible outside mission-critical domains. Additional research has aimed to provide fault protection via redundant multithreading [10, 22, 25, 28, 30]. Since processors which can execute multiple threads simultaneously are increasingly commonplace, the idea of using separate threads for error checking is a possibility. These techniques often require significant extra computations. Diva [1] is a less expensive alternative utilizing a small checker core to monitor computations performed by a larger microprocessor. Lower cost hardware checkers based solutions such as Argus [20] and others [7, 35] require small hardware changes. These hardware checkers based solutions still won't work for off-the-shelf hardware.

Embedded design spaces 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 technologies is providing just enough coverage (the percentage of errors that either get masked or can be detected and recovered from) of soft errors so that the effective fault rate remains at levels. Providing solutions which can achieve this coverage "on the cheap" is the goal of this work.

LCTES 2012, June 12–13, 2012, Beijing, China.

To achieve statistically significant soft error coverage at minimal overheads, we propose a software-only 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 [17, 32]. These symptoms can include memory access exceptions, divide-by-zero, and even mispredicted branches. At runtime, an individual symptom doesn't always signify a soft error, but a judicious use of these symptoms can be used to trigger a recovery. 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 [26, 27]. 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 coverage of more than 90%. However, the overheads in terms of performance and power are quite high since a large fraction of the application is replicated.

One of the key insights 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 detected 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 embedded microprocessors targeted for the consumer electronics market. However, unlike the high-availability IBM and HP servers that can provide provable guarantees on coverage, this work provides only opportunistic coverage, and is therefore not suitable for mission-critical applications. The contributions of this work are as follows:

- A software solution which does not need any user annotations in the application to generate reliability-aware code and works on applications written in a variety of languages.
- A selective instruction duplication approach that leverages memory profiling and edge profiling in compiler analysis to identify and replicate a small subset of vulnerable instructions not covered by symptom-based fault detection.
- Novel use of value profiling for the generation of software symptoms.
- Microarchitectural fault injection experiments to demonstrate the effectiveness of our proposed solution in terms of fault coverage and performance overhead.

2. Background and Motivation

2.1 Soft Error Rate (SER)

The effect of soft errors is becoming more pronounced as a result of transistor scaling. Aggressive scaling on one hand provides cheaper and more abundant transistors to pack on an individual chip, while on the other hand making each individual transistor more susceptible to soft errors. Traditionally, memory cells are more vulnerable to soft errors because they use smaller transistors to achieve higher densities and have inherent feedback mechanisms that can exacerbate the effect of small disturbances arising due to high energy particle strikes. Memory cells are mostly protected against soft errors by using parity checks or Error Correcting Codes (ECC). Due to shrinking device sizes for implementing logic in processors, the individual transistors in logic are also becoming vulnerable to soft errors. Additionally, combinational logic faults are harder to detect and correct. Shivakumar et al. [29] reported that the SER for SRAM cells is expected to remain stable, while the SER for logic is steadily rising. The aforementioned factors have motivated researchers to propose solutions to protect the microprocessor logic core against transient faults.

Feng et al. [9] and Shivakumar et al. [29] presented data for the effect of device 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.

Fortunately, around 75-92% of transient faults get masked (i.e., do not corrupt actual program state) due to architecture- or application-level masking. This masking can also occur at the circuit level. Our experiments show this masking rate to be around 78% collectively from all sources. Accounting for this masking, the raw SER for the present technology generation translates to about one failure every month in a population of 100 chips. For a typical commodity system such as laptop or mobile systems, this failure rate would be unnoticeable. However, in future technology 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 Instruction Duplication

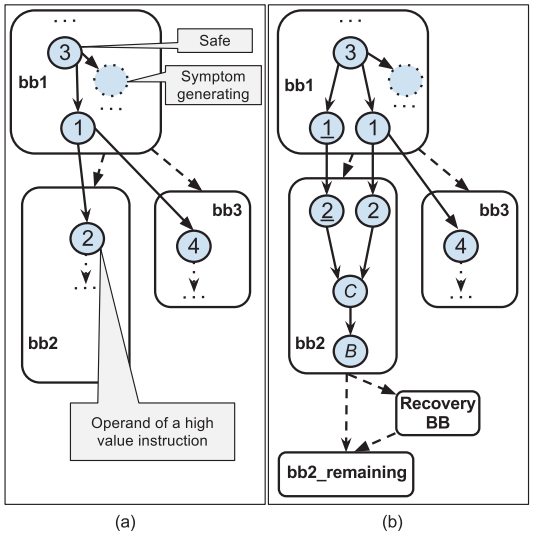

Figure 1: Duplicating instructions in a single thread of execution: Part (a) shows the original code and Part (b) shows the code after the duplicated instructions are inserted. Solid edges represent the data flow edges and dashed edges represent control flow edges. In (b), underlined nodes are duplicated nodes, and C and B nodes represent compare and branch instructions to compare the results from duplicated and original dataflow chains. The node with dashed outline is a symptom generating instruction.

In this Section, we provide an overview of the terminology used and point out the key differences with previously proposed instruction-duplication-based solutions. SWIFT [26] proposed the idea of duplicating instructions in a single thread of execution. The authors of SWIFT explain that a program has executed correctly if all the stores in the program have executed correctly assuming the program

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

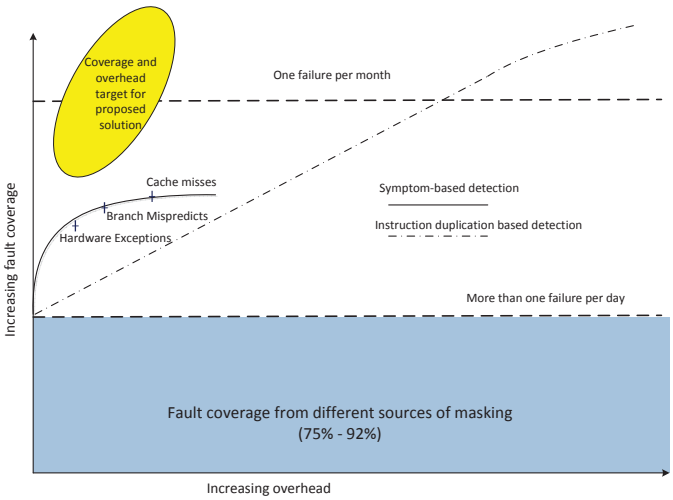

Figure 2: The trade-off between overhead and fault coverage from two existing fault detection schemes: symptom-based detection and instruction duplication-based detection. Also indicated is the region of the solution space targeted by our proposed technique. Our solution is aiming to provide between 90% and 99% coverage with little overhead. The dashed horizontal lines show user-visible failure rate for a single chip in a $16\,nm$ technology node with aggressive voltage scaling. This is a conceptual plot and is not to scale.

only communicates by writing data out through stores. Therefore, SWIFT recursively duplicated instructions by walking the data flow chains of the operands of stores and by protecting the control flow. Shoestring [9] 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, the terminology is adopted from Shoestring. The initial analysis phase of our solution classifies instructions into the categories described below.

- **Symptom-generating**: these instructions (e.g., address generation of loads and stores.) are likely to produce detectable symptoms if they consume a corrupted input.
- **High-value**: instructions (e.g., operands of I/O system calls.) which are likely to corrupt the output of the program if they consume a corrupted input.
- **Safe**: these instructions (e.g., those directly consumed by symptom-generating instructions.) are naturally covered by symptom-generating consumers.

Figure 1 shows the duplication process. Assuming node 2 is an operand of a high value instruction, the duplication starts at this node and walks the data flow chain until a safe instruction (node 3) is encountered. A duplicated instruction is placed just after the original instruction in program order. Compare and branch instructions are inserted to compare the results and to divert control flow to a recovery basic block. If the results match, the high value instruction is executed normally; Otherwise, recovery is triggered through the recovery basic block. In addition to encountering a safe instruction, the recursive duplication is terminated when 1) no more producers exist, and 2) the producers are already duplicated. Safe instructions are determined based on the probability of whether or not a particular instruction would generate a symptom if corrupted by a soft error.

2.3 Proposed Solution Landscape

As previously mentioned, a soft error solution that targets the commodity user space needs to be designed with lower overhead and acceptable coverage as targets. Figure 2 (data used from [9]) is a conceptual plot of overhead and coverage trade-off for 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. The bottom highlighted region in this plot indicates the amount of fault coverage that results from intrinsic sources of soft error masking, available naturally. The natural masking can occur because of many reasons such as register values being dead (i.e., such registers would be overwritten before they will be read) or Y-branches [33] (i.e., sometimes changing the direction of a conditional branch doesn't affect the correct program behavior). Among the remaining unmasked faults, symptom-based detection relies 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 faults, divide-by-zero, etc. If more frequently occurring microarchitectural events such as branch mispredicts and cache misses are included as symptoms, then recovery may be triggered more frequently, leading to an unacceptable amount of overhead [32]. In general, symptom-based methods provide good coverage at a relatively low overhead.

The coverage versus performance curve is far less steep for instruction duplication; The 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.

Figure 2 is generated in the context of a single $16\,nm$ chip with aggressive voltage scaling. The fault coverage provided by intrinsic sources of masking translates to more than one failure per day. This level of fault coverage is clearly unacceptable and might result in user visible corruptions very frequently. To achieve a more imperceptible failure rate, the fault coverage must be improved. Symptom-based and instruction-duplication methods combined can provide an acceptable level of coverage.

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. Figure 2 also shows the solution landscape targeted by our solution.

2.4 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 [11]. GCC [13] 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 profiling information we can exploit the common case behavior of a program to duplicate only those critical instructions. Different types of profiling information enables us to ignore unnecessary duplication of instructions that 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.

Specific details on different kinds of profile data used are presented in Section 3.

3. Proposed Solution

The main underlying observation behind our proposed solution is that 100% reliability is not always required. We need to keep the user visible corruptions at a level users have become accustomed to. Sensitive applications that are required to be executed reliably can be transformed with the compiler techniques developed as a part of the proposed solution. These applications will run marginally slower but will be able to tolerate more soft errors. Our proposed solution uses the idea of instruction duplication in a single thread of execution as explained in Section 2.2, and adds profiling-based intelligent tracing of dependences manifesting through memory to generate more efficient duplication code. In essence, our solution uses the dynamic behavior of applications to generate efficient code for transient fault detection.

3.1 Overview of proposed solution

Figure 3 shows our proposed solution framework in the context of machine-executable generation using the LLVM compiler framework [15]. The first step in this process is to convert the source code of the application to LLVM Intermediate Representation (IR, also called LLVM bit-code). In LLVM terminology, *passes* perform the transformations and optimizations that make up the compiler. Passes operating at the IR level either analyze the IR code or transform it from IR to IR, performing optimizations. Our duplication code framework is written as a pass in LLVM. The reliability-aware code generation pass analyzes and transforms the code by inserting duplicate instructions and comparisons as previously as described in Section 2.2.

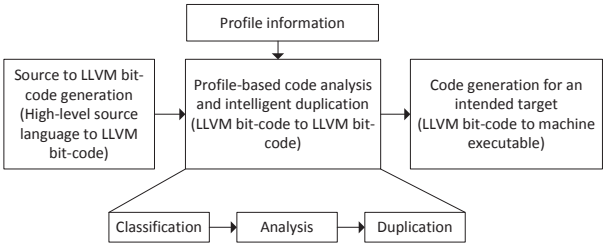

Figure 3: This Figure shows the flow of application compilation. LLVM bit-code is the internal representation of the LLVM compiler infrastructure. Our proposed solution operates at the LLVM bit-code level. Classification and analysis phases identify vulnerable parts of an application, and then the duplication phase protects the most vulnerable instructions by duplicating code.

An intuition behind our idea is that applications predominantly communicate to the external world using I/O library calls, and if we can capture the true input data flow chain of the operands of these calls, we can better protect the program output from getting corrupted. Under this observation, we can capture most, if not all, of the program I/O. This type of approach is suitable for our low overhead approach as we don't target 100% fault coverage. We include all library call and function call instructions as high-value instructions. An example where a program doesn't communicate using library calls is with the use of memory mapped I/O. An application might choose to memory map a file to communicate to the external world. Memory mapped locations can be used just like an array - direct loads and stores can be made to these memory locations. Using our technique, we can consider all stores as high value (at higher overhead) to protect applications with memory-mapped I/O.

We use LAMP [18], a toolset to trace and record the aliasing of memory addresses, to obtain memory profiling information. LAMP allows us to determine the data dependences that manifest through memory by reading and writing values at the same address. While duplicating instructions, our duplication algorithm walks the producer chain, considering the dependences through memory. In the recursive duplication of the producer chains of the operands of high value instructions, whenever a load is encountered, we consider the stores that aliased with the load and duplicate their producer chains too. By considering aliasing stores, the duplication algorithm of our solution achieves better and more useful code duplication. In our solution, the duplication process starts from the operands of library calls (high-value instructions). 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 that can corrupt program output. The producer chains of these stores are also protected by duplication. The remainder of this section describes the complete process from the analysis of the instructions to code duplication including the insertion of comparison instructions.

3.2 Overhead Reduction Without Losing Coverage

As mentioned previously, our solution detects soft errors by adding extra instructions in a single thread of execution, incurring a penalty in performance. In this section, we investigate techniques to reduce the overhead by using various kinds of profiling information. 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 further reduced 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 Simple Edge Profile based Pruning

The intuition behind this optimization is that frequently executed instructions should not be duplicated to protect an infrequently executed instruction. The probability of a soft error affecting an infrequently executed instruction is relatively low and so to protect such a instruction, unnecessary duplication of frequently executed instructions should not be performed. An example of this is shown in Figure 4. At the time of duplicating the instruction (node 4) in *bb3*, we check whether its operand-generating instruction (node 2) is executed frequently in comparison to the instruction itself. If this happens to be the case, the duplication is terminated for that particular data flow chain. If this optimization is used, then node 2 wouldn't be duplicated and as a result of this , we duplicate fewer instructions.

3.2.2 Using Memory Profiling Information

We use memory profiling to obtain information about aliasing between loads and stores. Also, memory profiling is used to identify silent stores that exist in an application. Further descriptions of these techniques follow.

Dependences Through Memory: As pointed out in Section 3.1, to duplicate the true dependences of the producer chains of high value instructions, we need load/store dependence information. 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. e.g., only library and function calls can be considered as high value instructions and only the operands of stores that alias with the loads in the producer chain of library call operands need to be protected.

Silent Store Optimization: A silent store is defined as a store that 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. [3] report 18% to 64% of total stores as silent for SPEC95 benchmarks. We have implemented silent store profiling as an extension of the LAMP toolset. 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 to our advantage.

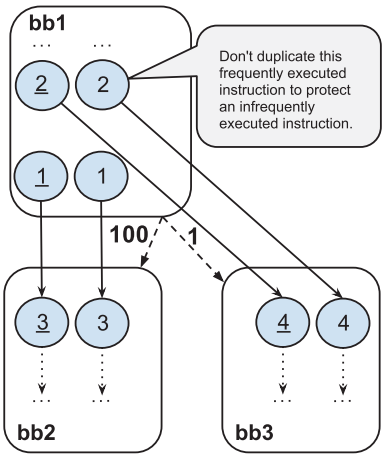

Figure 4: This Figure shows an example where execution frequency-based optimization is effective. The solid edges represent data flow edges and dashed edges represent control flow edges. Control flow edges are annotated with the execution frequency of the edge obtained using a profile run. Underlined numbers represent duplicated instructions. While duplicating an instruction in basic block *bb3*, if its operands' parent basic block is executed 100 times more frequently, then we don't duplicate its operand.

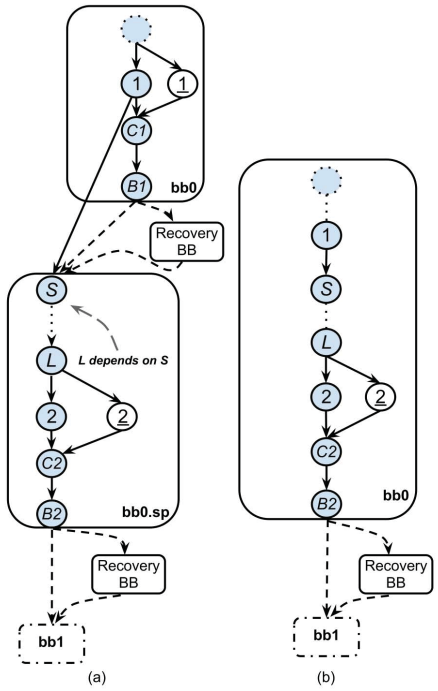

Figure 5: This Figure represents the control and data flow graphs for an example code. Solid arrows represent data flow edges and dashed edges represent control flow edges. In part (a), instructions 1 and 2 are both duplicated (seen underlined), with comparisons (C) and branches (B) to recovery code if a comparison fails. L represents a load instruction. If a silent store is on the path of the recursive producer chain, then the duplication process is terminated at that store and no source operands of the store are duplicated, as seen in part (b). The store instruction 'S' is assumed to be a silent store for this example.

For the purpose of this work, while doing recursive duplication, if we encounter a store which is almost always 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. The silent store removal optimization is explained in Figure 5 through an example. The duplication starts from the library call by walking the Data Flow Graph (DFG) and whenever a load is encountered, the recursive duplication continues with the operands of the stores that write to the same address as the load. Figure 5(a) shows duplication without considering the silent store optimization, and we end up duplicating more instructions. Figure 5(b) shows duplication when silent store optimization is enabled. If a store in the recursive duplication of a producer chain turns out to be silent, we terminate recursive duplication. This reduces the number of instructions duplicated. We use a threshold of 80% for a store to be considered silent since at runtime, it is not guaranteed that a store considered silent will always write the same value, and if a transient fault affects the store at such an execution instant, our technique will miss the fault. Such instances are expected to be rare because we choose a high threshold to classify a store to be silent.

3.3 Software Symptom Generation using Value Profiling

As mentioned in section 2.3, fault coverage that can be harnessed by using hardware symptoms saturates quickly (i.e., adding more symptoms doesn't improve fault coverage by a great extent). We have developed a novel value profiling-based method to generate software symptoms. If an instruction generates the same value almost 100% of the time, we can use that value and compare it 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 unintelligent duplication of the data flow chain. The benefits can only be seen in cases if the data flow chain is long and the count of instructions which would have been duplicated is greater than 2 (value cmp + branch instruction). In essence, this technique is expected to improve fault coverage by providing software symptoms and reduce overhead by a small amount.

An example where value profiling would be useful is provided in Figure 6. Figure 6(a) shows straight up duplication without considering value profiling. Say instruction 3 of Figure 6(a) generates the value '0' more than 99% of the time during the profiled execution of the program. While doing duplication by recursively traversing the operands, if instruction 3 is encountered in Figure 6(b) then an extra compare instruction is inserted to compare the value generated by it to '0'. If these two 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 instruction 3 produces output other than 0. Since this is 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

This work presents a solution to target soft errors induced by transient faults. The main cause of soft errors in microprocessors is high energy particle strikes. The experiments with high energy particle strikes conducted by Dixit et al. [8] 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

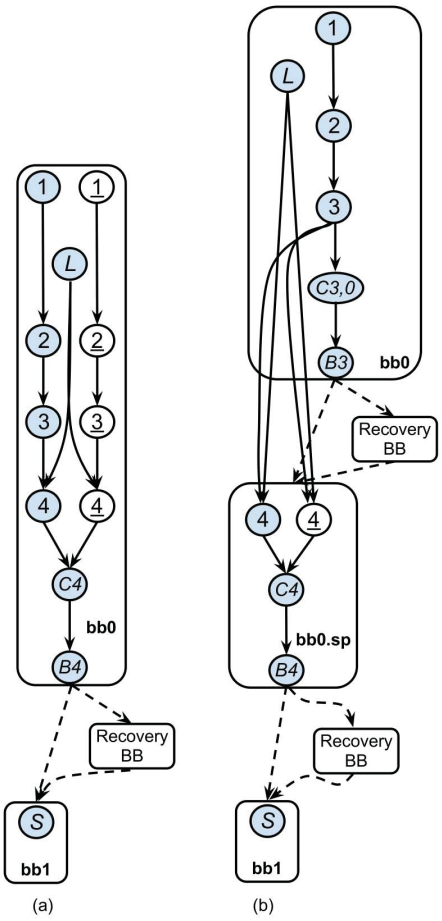

Figure 6: The effect of the value profiling on the instruction duplication process. Part (a) shows duplication without considering value profiling while part (b) shows duplication if value profiling is taken into account. Instruction 3 is assumed to generate the value '0' more than 99% of the time, and an extra comparison(C3,0) is added accordingly, jumping to additional recovery code if this comparison fails. Underlined instructions are duplicates, branches are indicated with 'B', and comparisons with 'C'.

errors. For the purpose of this work, we use a single bit-flip fault model implemented in the microarchitectural model of an 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 test 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 [15] 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. Some optimization passes such as machine common subexpression elimination can remove the duplicated instructions. We have disabled them during the phase when LLVM prepares the IR 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. If the LLVM bitcode is target independent, our code duplication framework can be used as-is to generate machine executable for a multitude of targets.

4.2 Fault Injection Framework

The fault model used in this work is a single bit-flip model. This model has been widely used in experimental evaluation of the previously proposed solutions to tackle the problem of soft errors. These faults are inserted by flipping a random bit at a random cycle during the course of application run. For the initial experiments, we injected faults randomly into the register file. In our experiments, faults in other microarchitectural structures are not explicitly injected, but faults in other structures predominantly manifest through register file as corrupted states. Thus, the register file is an attractive target for fault injection experiments. Wang et al. [34] showed that the bulk of transient fault-induced failures are dominated by corruptions introduced from injections into the register file. Overall, our technique is capable of detecting faults injected into other microarchitectural units that affect the program. Thus, injecting faults only into register file is a limitation of our evaluation infrastructure and is not a limitation of our proposed technique. For the purpose of this work, we have used the GEM5 [5] simulator. The simulator was run in ARM syscall emulation mode and modeled the ARMv7-a profile of ARM architecture. We have used a model of the in-order ARM architecture. Since our injection site is the register file, we expect that an out-of-order model wouldn't affect our conclusions significantly. In fact, we believe that an out-of-order model will improve our results 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. Fault injection experiments with an out-of-order implementation are planned as a part of our future work. The details of the processor configuration used for the experiments are in Table 1.

Table 1: GEM5 Simulator parameters (models an ARMv7-a profile of ARM architecture).

Processor core @ 2GHz	
Simulates an In-order core	
Physical register file size	16 entries
Simulation Mode	Syscall Emulation
Memory	
L1-I/L1-D cache	32KB, 2-way
L2 cache (unified)	2MB, 16-way
DTLB/ITLB	64 entries(each)

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. We have only used user mode registers to inject faults. Injecting faults in privileged mode registers would yield a higher masking rate because no benchmarks use these registers and so, injected faults would have no effect. To stress test our technique, we chose to ignore injecting faults in privileged mode registers and as a result, a lower masking rate is observed in comparison to the masking rate reported in previous research [34] efforts with soft errors.

To calculate the statistical significance of a given number of fault injection trials, we use the works of Leveugle et al. [16]. We need 96 fault injection trials for each benchmark to have a 10% margin of error

and confidence level of 95%. Ideally, we would like to perform our experiments with a 5% margin of error and a confidence level of 95% but this amounts to 384 trials per benchmark. Considering we have 10 benchmarks and we need perform fault injection experiments for full duplication, the baseline, and our proposed technique, running 384 trials per benchmark would lead to a very long simulation time. The approximate time would be 23040 (3*10*384*2) hours of simulation assuming 2 hours of average runtime for each benchmark. Therefore, we chose 100 fault injection trials for each benchmark to yield results with reasonable accuracy in a timely manner. After the fault injection, the program runs until completion and the log files are collected. 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 so that a recovery can be triggered. The next section 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. **Silent corruptions or infinite loop**: Faults that produce user visible corruptions, cause early program termination, or do not terminate in definite time 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. From an architecture perspective, this idea of failure may seem inaccurate, but it is consistent with recent symptom-based work and is the most appropriate in the context of evaluating our current work. The main motivation behind our solution is that 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.

4.3 Recovery Support

Our solution relies on the ability to roll back processor state to a clean checkpoint. Wang and Patel [32] indicate that checkpointing and recovery are possible if the fault can be detected within a window of 1000 instructions for speculated pipelines. 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.

4.4 Benchmarks

We have used 10 applications from the SPECINT2000 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. In this work, multithreaded programs are not considered. However, we do not foresee any problems of using our technique with race-free multithreaded programs. Code duplication in a multithreaded 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. In the context of embedded systems if the change in execution time affects program output, these programs might not run correctly after partial duplication. Experiments with multithreaded programs are left as an interesting direction to explore further.

5. Experimental Results

In this section, the effectiveness of various techniques presented in this work is analyzed using the experimental setup described earlier. First, the data for silent stores is presented. We then analyze the maximum amount of fault coverage we can obtain from full duplication. Finally, the effect of using memory profiling for tracing dependences through memory is analyzed in comparison to previous works.

5.1 Silent Stores

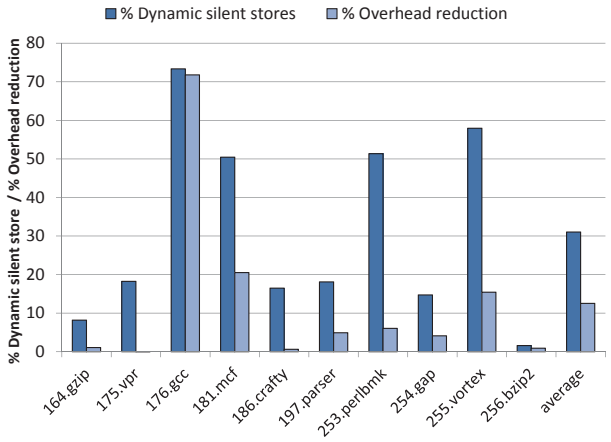

Figure 7: The % Dynamic silent stores bar shows dynamic silent stores as a percentage of total dynamic stores in a benchmark. The high percentage of silent stores in some benchmarks suggest that their presence can be exploited for intelligent code duplication. The % Overhead reduction bar shows the reduction in performance overhead if silent store optimization is used while duplicating instructions. Notice that the benchmarks showing a large percentage of silent stores also show a significant reduction in overhead.

The % Dynamic silent stores column in Figure 7 shows the number of dynamic silent stores as a percentage of total stores for various applications. 176.gcc, 181.mcf, 253.perlbmk and 255.vortex show a high percentage of dynamic silent stores and these also show a significant reduction in overhead as shown in the % Overhead Reduction column in Figure 7. For the results presented in Figure 7, duplication is terminated (see Section 3.2.2) 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 show less 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 previous works using the criteria of performance overhead and fault coverage. If a fault results in masking, SWDetect or symptoms, system can correctly execute the program. Hence, fault coverage is defined as the percentage of injected faults that result in masking, SWDetect or symptoms.

In this first experiment, we examine the maximum amount of coverage we can obtain by doing the maximum amount of duplication. Since loads are never duplicated to save on memory traffic, the overhead wouldn't be 100% for full duplication and there will always be some faults which can escape detection by the duplicated code. The full duplication column in Figure 8 shows the performance overhead if the duplication is not terminated at safe instructions and all the branches are also protected by duplication. The full-dup column in Figure 9 is the corresponding fault coverage breakdown among the

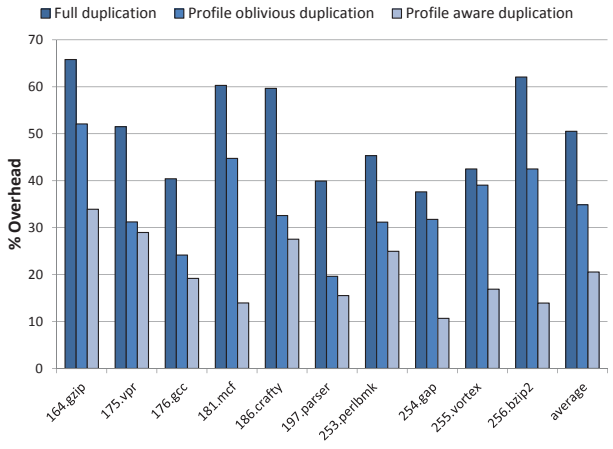

Figure 8: Overhead comparison among full duplication, profile oblivious duplication, and profile aware duplication. In full duplication, duplication is not terminated at safe instructions and all branches are also protected. Although profile oblivious duplication uses safe instructions, profiling information is not utilized. This represents a system equivalent to Shoestring. Profile-aware duplication uses safe instructions as well as profiling information.

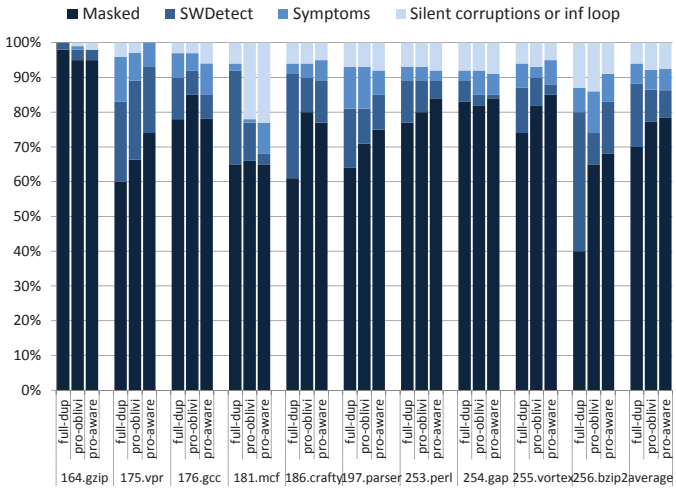

Figure 9: Coverage breakdown for full duplication (full-dup), profile oblivious duplication (pro-oblivi) and profile aware duplication (pro-aware).

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 94%. The performance overheads in this Section are compared to -O3 optimized baseline. 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 8 and pro-oblivi column in Figure 9 show the performance overhead and fault coverage numbers if the duplication is terminated at safe instructions and only the immediate branch whose execution affects the execution of high value instruction is protected by duplication. This is equivalent to the Shoestring solution. It reduces overhead but fault coverage deceases from 94% to 92.2%. For the rest of results, we have considered profile oblivious duplication as our baseline values for result comparisons.

A general trend observed in the results is that with lesser duplication, masking goes up. For example, profile oblivious duplication (pro-oblivi) has lower overhead than full duplication (full-dup) on average (Figure 8), hence lesser duplication, but has more masking than full-dup (Figure 9). This stems from the fact that with less duplication, there a decreased chance of fault detection and therefore a greater chance of fault masking or overall failure since undetected faults result in masking or failure. Since the amount of duplication in an application changes its code structure, randomly injected faults in the same application with different levels of duplication show different behavior.

The profile-aware duplication column in Figure 8 shows the overhead if we duplicate the producer chains of library and function calls only (i.e., only library and function calls are considered as high value instructions) and make use of profile information. The pro-aware column in Figure 9 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. Our technique results in better code duplication, providing the same level of fault coverage seen with our baseline but at 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 presented. We have combined the contributions of edge profiling and silent store optimization together and the results in this section are presented for a subset of benchmarks because running 100 fault injection trials for each configuration leads to a large number of simulations. These benchmarks are not handpicked because they show desirable behavior.

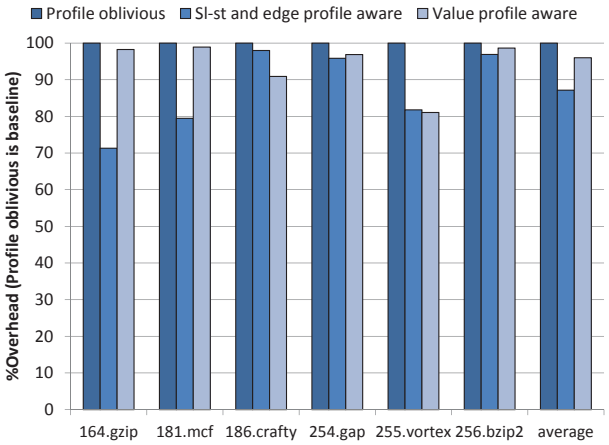

Figure 10: The profile-oblivious column is the baseline overhead. The reduction in overhead if we use the silent store optimization and edge profiling information is shown in the 'Sl-st and edge profile aware' column. The value profile aware column shows the reduction in overhead if we use value profile in comparison to our baseline.

The 'Sl-st and edge profile aware' column in Figure 10 show the reduction in overhead if the silent store and edge profile based optimizations are used. The profile oblivious duplication bar is the baseline overhead. In comparison to our baseline, these two techniques combined result in a 12.78% reduction in overhead. The sl-edge-aware column in Figure 11 shows the coverage breakdown among

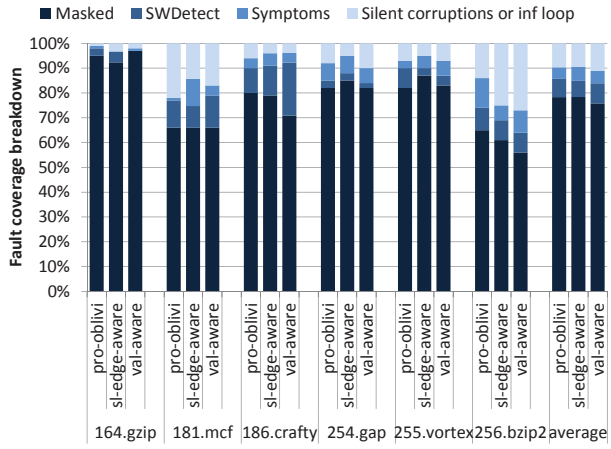

Figure 11: The pro-oblivi column shows the coverage breakdown for our baseline . The coverage breakdown if we use silent store optimization and edge profile information is shown in the sl-edge-aware column. The val-aware column shows the coverage breakdown for value profile aware code duplication.

different components. On average, because of software duplication, the combined fault coverage stays the same. As shown in the val-aware column in Figure 10, the use of value profiling provides a 5.9% reduction in the performance overhead of duplication on average. Value profiling provides a slight increase in the number of faults covered by duplication while reducing the overhead.

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.

6. Related Work

This section describes the work that is related to our proposed solution. Software instruction duplication is an approach which is extended in our work in an effort to increase fault-coverage while reducing performance 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. [26]. The previous works in software-based instruction duplication are [9, 26], the most closely-related works to our solution. Our work differs from these works in the following ways:

- Our work makes novel use of value profiling to generate extra software-based symptoms.
- SWIFT [26] considered all the stores as starting point for duplication. Shoestring [9] improved upon that by considering global stores and all functions calls as starting point for instruction duplication. Our solution starts duplicating instructions only from library and function calls and then uses memory profiling to find the true load/store dependencies. In this process, only the important stores get considered as high value and a lesser duplication overhead is achieved.
- Silent store profiling information is incorporated in this work for the first time.
- Unlike some of the previous works, our solution is not tied to a specific ISA. We take a fresh approach, and instruction duplication 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.

Other works such as CRAFT and PROFIT [27] improve upon the SWIFT solution by leveraging additional hardware structures and architectural vulnerability factor (AVF) analysis [23], respectively. Compiler-based instruction duplication delivers nearly complete fault coverage, with the added benefit of requiring little to no hardware cost. However, in order to achieve this, solutions like SWIFT can more than double the number of dynamic instructions for a program, incurring significant performance and power penalties which are costly to implement in embedded devices. Latif et al. [14] present a software based solution which exploits data representation for fault detection. It doesn't handle arbitrary C/C++ programs.

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

Some recent solutions have also suggested the idea of distributed checking in the core for various components. Argus [20], for example, relies on a series of hardware checker units to perform online invariant checking to ensure correct application execution. Our solution differs from all of these techniques because it does not require any special hardware modifications.

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 [32], 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. The strength of symptom-based detection lies in its low cost and ease of application. mSWAT [12] presented a solution which detects anomalous software behavior to provide a reliable system. It requires special simple hardware detectors to detect faults.

One final approach to soft error tolerance targets another aspect of the microarchitecture, the register file. Register file protection schemes are based on the premise that faults occurring in the register file are statistically more likely to corrupt the output of the program. As ECC is applied to main memory to protect against soft errors, the same technique can also be applied to the register file. Solutions like the one presented by Montesinos et al. [21] build upon this insight and only maintain ECC for those registers most likely to contain live values. ECC protection would only be helpful if the soft error corrupts a register after it has been written; If faulty data gets written to registers, ECC is simply useless. In contrast, our solution can detect errors which occur elsewhere in the architecture but propagate to the register file. Similarly, Blome et al. [6] propose a register value cache that holds duplicates of live register values to aid in the protection process.

7. Conclusions and Future Work

The relentless 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. We make novel use of value profiling for generating software symptoms. In this work, we presented a solution that uses profile-based compiler analysis to selectively duplicate instructions. Our profile based selective duplication results in a reduction of overhead of 41% in comparison to a previously proposed solution while maintaining the same level of fault coverage.

Future work includes focusing on identifying more opportunities to reduce performance overhead without affecting transient fault coverage. We also plan to extend the simulator infrastructure to inject faults in other microarchitecture structures such as the decoder, branch predictor, TLB, etc. to observe the effect on fault coverage of our currently proposed solution. We also wish to validate the technique pre-

sented in this work on multithreaded programs, and on an out-of-order infrastructure.

8. Acknowledgement

We thank the anonymous reviewers for their constructive comments and suggestions for improving the work. This research was supported by the National Science Foundation under grant CCF-0916689 and the Toyota InfoTechnology Center.

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] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

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

[9] 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.

[10] 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.

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

[12] S. Hari, M.-L. Li, P. Ramachandran, B. Choi, and S. Adve. mswat: Low-cost hardware fault detection and diagnosis for multicore systems. In *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on*, pages 122 –132, dec. 2009.

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

[14] M. M. Latif, R. Ramaseshan, and F. Mueller. Soft error protection via fault-resilient data representations. In *Workshop on Silicon Errors in Logic - System Effects*, 2007.

[15] 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.

[16] 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.

[17] 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.

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

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

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

[21] 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.

[22] 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.

[23] 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.

[24] E. Normand. Single event upset at ground level. *Nuclear Science, IEEE Transactions on*, 43(6):2742 –2750, dec 1996. ISSN 0018-9499. doi: 10.1109/23.556861.

[25] 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.

[26] 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.

[27] 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.

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

[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] 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.

[32] 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.

[33] 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.

[34] 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.

[35] 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.

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