Comprehensive Failure Analysis against Soft Errors from Hardware and Software Perspectives

Yohan Ko*, Hwisoo So†, Jinhyo Jung†, Kyoungwoo Lee†, and Aviral Shrivastava‡
*Division of Software, Yonsei University, 1 Yonseidae-gil, Gangwon-do, 26493, Republic of Korea
†Yonsei University, 50 Yonsei-ro, Seodaemun-gu, Seoul, 03722, Republic of Korea
‡Arizona State University, Centerpoint, 660, S Mill Ave, Tempe, AZ, 85281, United States

***Abstract*—With technology scaling, reliability against soft errors is becoming an important design concern for modern embedded systems. To avoid the high cost and performance overheads of full protection techniques, several researches have therefore turned their focus to selective protection techniques. This increases the need to accurately identify the most vulnerable components or instructions in a system. In this paper, we analyze the vulnerability of a system from both the hardware and software perspectives through intensive fault injection trials. From the hardware perspective, we find the most vulnerable hardware components by calculating component-wise failure rates. From the software perspective, we identify the most vulnerable instructions by using the novel root cause instruction analysis. With our results, we show that it is possible to reduce the failure rate of a system to only 12.40% with minimal protection.**

***Index Terms*—Soft Error, Transient Fault, Fault Injection, Failure Analysis, Reliability**

I. Introduction

Soft errors, or transient faults, are caused by external radiation such as alpha particles, thermal neutrons, and cosmic rays [1]. If the radiation-induced charge is larger than a certain threshold, known as the critical charge, a soft error can occur and cause bit flips in the hardware, leading to timeouts, system failures, or incorrect outputs. Thus, reliability against soft errors is an essential concern in modern embedded systems [2].

Several techniques have been presented to protect embedded systems against soft errors. Example techniques include information redundancy schemes [3] and software instruction duplication [4]. However, these protection techniques are inappropriate for resource-constrained embedded systems since they are expensive or inefficient. In order to mitigate protection overheads, selective or partial protections have been proposed [5] [6]. However, partial protection methods relied mostly on heuristics to find the critical parts of the system.

In this work, we have conducted a comprehensive study to find the microarchitectural components and software instructions that are most vulnerable to soft errors. We modified the cycle-accurate system-level gem5 simulator [7] to perform extensive fault injection campaigns into benchmarks from the MiBench [8] benchmark suite. From the hardware perspective, we injected faults into specific microarchitectural components to estimate the vulnerability of each component separately. From the software perspective, we performed the root cause instruction analysis, a novel technique to find the software instruction responsible for the system failure. Based on the results of our experiments, we present protection guidelines for resource-constrained embedded systems.

II. Failure Analysis Framework

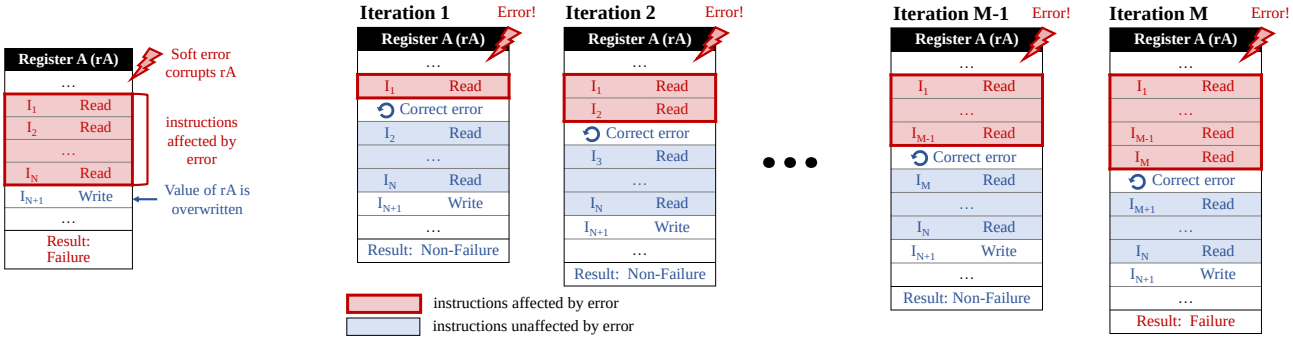
To study the vulnerabilities of microarchitectural components and instructions in a comprehensive manner, we have developed a failure analysis framework based on the cycle-accurate gem5 simulator [7]. Our framework includes a fault injection module to inject single-bit soft errors to an in-order CPU. The single-bit error is injected to a randomly selected bit at a randomly selected cycle within the execution time in one of four main components (register file, pipeline register, LSQ, and scoreboard) excluding the cache. Then, our framework runs several iterations of a cross-compiled benchmark with the fault injections and returns simulation statistics, program outputs, and simulation trace which is composed of hardware-level microarchitectural behaviors and software-level instructions.

We used our framework to perform extensive fault injection experiments over various benchmarks and calculated the vulnerability of microarchitectural components and software instructions. We also used the trace data and logs from the injection trials for further analyses. If a trial resulted in a failure, the information was used to pinpoint the root cause of the failure. Otherwise, we analyzed how the injected fault was masked from the system's perspective. Thus, our framework can act as a toolset for comprehensive investigation on how soft errors impact the system from the hardware and software perspectives.

A. Failures in Hardware Components

Injection-based failure analysis from the hardware perspective is relatively simple since we can control in which microarchitectural component to inject the fault. We perform our analysis in the register file, pipeline register, LSQ, and scoreboard. We do not inject faults into the cache assuming that the cache is protected with parity or ECC [6]. otherwise, the cache would be far more vulnerable than other components due to its size and density, and we feared that the vulnerability of the cache would overshadow the differences between the vulnerabilities of other components.

An error in data caused by an injection propagates when it is read by the processor. Therefore, our framework keeps track of the reads and writes on the injected bit after the cycle the



(a) **First-level analysis**: Find all instructions that read corrupted data. In this case, $I_1 \sim I_N$ are the candidate root cause instructions.

(b) **Second-level analysis**: Correct the corrupted data after m instructions access the value. m starts from 1 and is incremented every iteration until system failure occurs. In this case, I_M is the root cause instruction since it is the first instruction to induce system failures.

Fig. 1. Two-level analysis to identify the root cause instructions when faults are injected into the register file

fault was injected. For example, consider the instruction "load r1, r2", which loads the data in the memory address designated by register r2 to register r1. Injections in the register file leads to one of two cases. If the data in r2 becomes corrupted, the processor may access the wrong data or an invalid memory segment. If the data in r1 becomes corrupted, the error is overwritten to a correct value and becomes masked. In this case, our framework ceases to track the reads and writes on the corrupted bit. Injections in the LSQ lead to similar results. When the exemplary instruction "load r1, r2" is executed, the memory address is first inserted into the LSQ by reading the data in r2. Then, the LSQ accesses the data cache to load the data designated by the memory address, and the data is inserted into LSQ. Finally, r1 is updated using the memory data in LSQ. Injections in the memory address before the LSQ accesses the data cache could lead to incorrect memory access. Injections in the memory data before the update to r1 would corrupt the value of r1. Now consider injections into the pipeline register. If the injected fault corrupts the opcode, the "load" instruction could become any other type of instruction and induce failures. If one of the operands is changed due to the fault injection, the instruction will read data from an incorrect register (e.g., load r1, *r3*) or refer to an incorrect register as the destination (e.g., load *r3*, r2). Finally, consider the case of the scoreboard. Before the instruction "load r1, r2" is executed, the destination register index (r1) is logged in the scoreboard to prevent any violations of data dependency. If the fault causes the scoreboard to log r0 instead of r1, subsequent instructions using r1 may not wait for the load instruction to complete and read the incorrect data of r1 before the update.

B. Failures in Software Instructions

Failure analysis of software instructions is more complicated than that of hardware components. In injection experiments, faults are injected into hardware components to imitate the behaviors of soft errors in the real world. An additional step must be taken to find the software instructions that are affected by the injected fault. In some cases, more than one instruction may access the injected bit, further complicating the issue. We, therefore, require a method that first finds instructions affected by the fault, then pinpoints the single instruction responsible for the failure. This is possible through the novel root cause instruction analysis technique, which identifies the first instruction that eventually causes a system failure.

Our framework follows a two-level analysis to identify the root cause instruction, as shown in Fig. 1. Assume that the data in register A (rA) becomes corrupted, and the program results in a system failure. In the first-level analysis, our framework finds all the candidates of the root cause instruction, which are the instructions that read data from the corrupted register. If there is only one instruction, the instruction is designated as the root cause instruction, and the second-level analysis can be skipped. In the other case where multiple instructions read the faulty data, we apply the second-level analysis to find the root cause instruction.

The core idea of the second-level analysis is to isolate the effect of the error to one instruction at a time. This involves multiple iterations of the program with an error correction function, which we have implemented in our framework. To illustrate a sample run of the second-level analysis, assume that a failure-causing fault is injected into register A (rA) and is read by N instructions, as shown in Fig. 1(a). In the first iteration, we correct the value of rA immediately after the first instruction (I_1) uses the value. This way, the effect of the error is isolated to only I_1. If the iteration does not result in a system failure despite I_1 reading the wrong value, then I_1 is not the root cause instruction. In this case, we proceed to the second iteration, in which the error is corrected after the execution of the second instruction. We repeat this process of incrementing the number of instructions that read the erroneous value of rA until an iteration results in a system failure. Then, the instruction executed immediately before the correction is responsible for the failure and is labeled as the root cause instruction. This process is depicted in Fig. 1(b).

With the novel root cause instruction analysis, each injected fault that causes system failures can be traced back to the instruction responsible for the failure. Then, the injected faults can be abstracted out to the software instruction level

without worrying about the details of the underlying hardware. This software perspective analysis can show what kind of instructions are more vulnerable in each benchmark and explain which instructions should be protected when selective protection techniques are applied.

III. Failure Analysis and Protection Guidelines

Using our failure analysis framework, we have performed extensive fault injection campaigns targeting a 32-bit ARM architecture. For ease of failure analysis, the scope of our experiments only covers single-bit errors. We do not consider multiple-bit upsets since the rate of double-bit errors is only 1/100 of the single-bit error rate [9]. Our fault injection campaigns cover 7 benchmarks (*matmul*, *stringsearch*, *gsm*, *susan*, *jpeg*, *qsort* and *bitcount*) from the MiBench [8] benchmark suite, with 10,000 injections per component for each benchmark, one injection per trial. From these trials, the failure rate is calculated as $Failure\ Rate = \frac{Number\ of\ Failure\ Cases}{Total\ Number\ of\ Trials}$.

From the hardware perspective, the pipeline register is the most vulnerable microarchitectural component with over 63% failure rate, compared to the 40%, 31%, and 5%, of the register file, LSQ, and scoreboard, respectively. This is mainly because the pipeline register contains information crucial for correct execution, such as the operations and operands. To make things worse, the pipeline register is filled with data to be used, which makes it experience much less masking compared to other components. On the other hand, the scoreboard is the least vulnerable since faults in the scoreboard only affect data dependency between instructions. If the register indicated by the injected bit shares no data dependency with other instructions, the corrupted data in the scoreboard is not used, and the error is immediately masked.

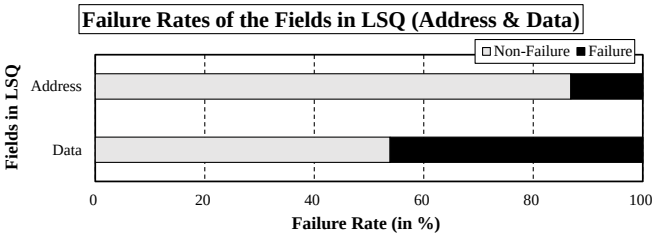


Fig. 2. Detailed failure analysis of the LSQ. The chances of failures are much higher when faults are injected in the memory data as opposed to the address.

We take a closer look into the LSQ in Fig. 2 to investigate its relatively low failure rate. In this graph, the x-axis represents the rate of system failures, and the y-axis shows where the fault was injected: "Address" meaning that faults were injected into the memory, "Data" meaning that faults were injected into the data. Interestingly, the failure rate of the memory data is almost 4 times higher than the failure rate of the memory address (46% compared to 13%). The reason for the low failure rate of the memory address, and in turn the low failure rate of LSQ as a whole, is due to the mechanism of the LSQ. For load instructions, the memory data portion of the LSQ is non-vulnerable until the data is loaded from memory since the load will overwrite any errors that may have occurred. After the memory access is made, the memory address is no longer needed and becomes non-vulnerable. The memory data and address for stores also become non-vulnerable as soon as the memory access is complete. Therefore, in contrast to other memory components, the bits in the LSQ are vulnerable for only a limited amount of time. Then the bits that become non-vulnerable shield the rest of the LSQ through spatial masking, reducing the failure rate of LSQ to about 30%.

From the software-perspective, we first categorize software instructions into six types: load, store, arithmetic, logical, compare, and branch. We then compare the failure rates of each type of the root cause instruction. Memory instructions show relatively low failure rates of about 30%(load) and 40%(store). This result agrees with the hardware perspective analysis, which showed that LSQ had low failure rates due to spatial masking. On the other hand, control flow instructions have higher failure rates. Branch instructions are the most vulnerable with 55% failure rates, and compare instructions, whose results are most often used as parameters for branch instructions, also show high failure rates of 47%. While errors in other instructions change the output of one instruction in most cases, incorrect execution of branch instruction can corrupt the execution flow. Instead of producing one incorrect value, errors in control flow instructions produce several incorrect operations, raising the chances of the error propagating to system failures.

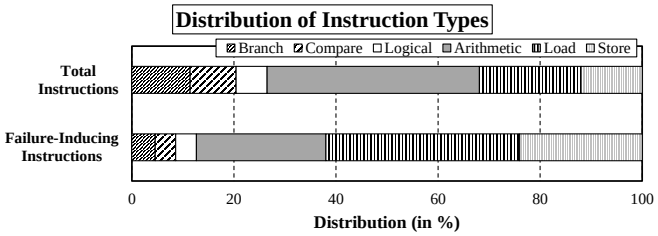


Fig. 3. Distribution of different instruction types. Load instructions take up about 20% of the system, but are responsible for almost 40% of the failures.

However, the distribution of failure-inducing root cause instructions suggests otherwise. Fig. 3 summarizes the distribution of instruction types in the system overall and in failure-inducing root cause instructions. Branch instructions and compare instructions, which showed high failure rates of 55% and 47%, are responsible for only 4.5% and 4.0% of the failures respectively. On the other hand, 37.9% of failures are caused by load instructions, which showed lower failure rates of about 30%. This is not only because load instructions occur much more frequently than branch or compare instructions. Load instructions take up 19.9% of the total instructions, which is slightly less than the 20.4% of the branch and compare instructions combined. This implies that load instructions are much more likely to become root cause instructions, and should be protected.

In conclusion, it is generally most efficient to apply hardware protection on the pipeline register. The pipeline register

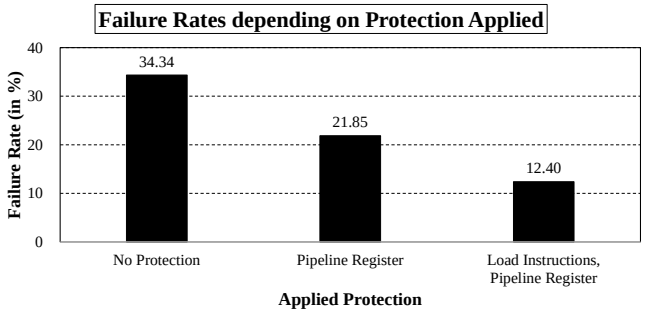


Fig. 4. By protecting one hardware component and instruction type, the failure rate decreases to 12.40%.

has high failure rates since it holds information crucial to correct execution and holds almost no useless information. From the software perspective, load instructions are typically the most vulnerable due to their dominance in the distribution of failures. Therefore, by applying a hybrid selective protection technique to protect the pipeline register and load instructions, the overall failure rate drops to 12.40%, as shown in Fig. 4.

IV. Related Works

Several techniques have been proposed to protect processors against soft errors. Early attempts aimed to protect the whole system. Examples include shielding the system from external radiation with a concrete barrier [10], or duplicating all software instructions and comparing the results between original and duplicated instructions to detect soft errors [4]. However, the area, cost, and performance overheads induced by these methods are intolerable for most embedded systems. Many studies have therefore turned to selective protection. From the hardware perspective, Naseer et al. [11] suggest that the register file could be the most vulnerable since errors in the register file can quickly and easily propagate to other components. Other candidates include the pipeline register [5], and the scoreboard [12]. From the software perspective, Reis et al. [6] saw store instructions as the most critical. Control flow instructions are also considered vulnerable since an incorrect control flow can execute incorrect store instructions or omit the execution of correct store instructions.

Measuring the reliability of a system is another field of study. One notable method is neutron beam testing [13], which exposes the target processor to neutron-induced soft errors. Beam testing experiments are considered highly accurate because the experimental environment is very similar to how soft errors actually occur. On the downside, they are expensive to perform and hard to set up correctly. Fault injection campaigns have been presented as an alternative to the expensive and complicated beam testing [14]. Ideally, the failure rate could be measured by injecting soft errors to all bits in each cycle. As this is near impossible, most works adopt the idea of statistical fault injection [15]. The idea is to perform a number of trials large enough to make the results statistically significant, but much smaller than the number required by exhaustive fault injection ($number\ \ of\ \ bits \times cycles$). For example, a statistical fault injection experiment may involve 10,000 injections. According to probability theory, this number is sufficient to achieve 1% margin of error with 95% confidence level, regardless of the population size.

V. Conclusion

Soft errors are important reliability issues at the early design phase, but protections against soft errors incur severe overheads in terms of hardware area and performance. Selective protection techniques have been proposed for resource-constrained embedded systems, but their resilience needs to be validated. In this study, we use our framework with our novel root cause instruction analysis to analyze the most vulnerable parts of a system from both hardware and software perspectives. From the hardware perspective, protecting the pipeline register is the most essential, and from the software perspective, protecting the load instructions is the most efficient. Using these ideas, we reduce the failure rate of an application by over 60%(from 34.34% to 12.40%) on average, just by protecting the pipeline register and load instructions.

VI. Acknowledgments

This work was partially supported by funding from National Science Foundation Grants No. CNS 1525855, CPS 1646235, CCF 1723476 - the NSF/Intel joint research center for Computer Assisted Programming for Heterogeneous Architectures (CAPA), 2014-3-00035 (High Performance and Scalable Manycore Operating System, IITP, MSIT), and Samsung Electronics Co., Ltd(FOUNDRY-202108DD007F).

References

[1] N. Seifert *et al.*, "Soft error susceptibilities of 22 nm tri-gate devices," *IEEE Transactions on Nuclear Science*, vol. 59, no. 6, 2012.
[2] V. Narayanan and Y. Xie, "Reliability concerns in embedded system designs," *Computer*, vol. 39, no. 1, pp. 118–120, 2006.
[3] R. Baumann, "Soft errors in advanced computer systems," *IEEE Design & Test of Computers*, vol. 22, no. 3, pp. 258–266, 2005.
[4] N. Oh *et al.*, "Error detection by duplicated instructions in super-scalar processors," *IEEE Transactions on Reliability*, vol. 51, no. 1, 2002.
[5] R. Jeyapaul *et al.*, "Systematic methodology for the quantitative analysis of pipeline-register reliability," *IEEE Transactions on VLSI Systems*, vol. 25, no. 2, 2016.
[6] G. A. Reis *et al.*, "SWIFT: Software implemented fault tolerance," in *CGO*, 2005, pp. 243–254.
[7] N. Binkert *et al.*, "The gem5 simulator," *SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, 2011.
[8] M. R. Guthaus *et al.*, "MiBench: A free, commercially representative embedded benchmark suite," in *WWC*, 2001, pp. 3–14.
[9] K. Lee *et al.*, "Mitigating the impact of hardware defects on multimedia applications: A cross-layer approach," in *MM*, 2008, pp. 319–328.
[10] A. Lesea *et al.*, "The rosetta experiment: atmospheric soft error rate testing in differing technology FPGAs," *IEEE Transactions on Device and Materials Reliability*, vol. 5, no. 3, pp. 317–328, 2005.
[11] R. Naseer *et al.*, "Analysis of soft error mitigation techniques for register files in IBM Cu-08 90nm technology," in *MWSCAS*, vol. 1, 2006.
[12] Monferrer *et al.*, "Mechanism for soft error detection and recovery in issue queues," 2007, uS Patent App. 11/999,787.
[13] A. Dixit and A. Wood, "The impact of new technology on soft error rates," in *IRPS*, 2011, pp. 5B.4.1–5B.4.7.
[14] K. Parasyris *et al.*, "GemFI: A fault injection tool for studying the behavior of applications on unreliable substrates," in *DSN*, 2014.
[15] R. Leveugle *et al.*, "Statistical fault injection: Quantified error and confidence," in *DATE*, 2009, pp. 502–506.