Root cause analysis of soft-error-induced failures from hardware and software perspectives

Jinhyo Jung[a], Yohan Ko[b,*], Hwisoo So[a], Kyoungwoo Lee[a] and Aviral Shrivastava[c]

[a] *Yonsei University, 50 Yonsei-ro, Seodaemun-gu, Seoul, 03722, Republic Of Korea*
[b] *Yonsei University, 1 Yonseidae-gil, Gangwon-do, 26493, Republic Of Korea*
[c] *Arizona State University, 660 S Mill Ave, Tempe, AZ 85281, United States*

ARTICLE INFO

Keywords:
Soft Error
Transient Fault
Fault Injection
Failure Analysis
Reliability

ABSTRACT

Because the dangers of soft errors are increasing with continued technology scaling, reliability against soft errors is becoming an important design concern for modern embedded systems. Various schemes have been proposed to protect embedded systems from the threat of soft errors, but they incur considerable overheads in terms of cost and performance. Selective protection techniques seem promising because they can achieve high levels of protection with low overhead. Though these techniques can be applied to any system, the most vulnerable parts must first be identified. We, therefore, present CFA, a comprehensive failure analysis framework that can analyze the vulnerability of microarchitectural components and software instructions through intensive fault injection campaigns. With CFA, we also explore the vulnerability of ten benchmarks from the MiBench benchmark suite. We found that protecting a part of the system heavily affects the reliability of the other parts. Therefore, all combinations of protection methods must be examined to present the most efficient and effective protection guidelines. Throughout the experiments, we observed that protection methods offered by single-perspective analyses are sub-optimal. On the other hand, CFA finds the optimal solution in every case, reducing the AVF of a system by up to 82% with minimal protection.

1. Introduction

Soft errors, or transient faults, are caused by external radiation such as alpha particles, thermal neutrons, cross-talks, and cosmic rays [36]. If the radiation-induced charge exceeds 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. The critical charge of contemporary devices is continuously decreasing owing to the reduction in chip size and supply voltage, and accordingly, soft error rates are exponentially increasing [2, 25]. Thus, reliability against soft errors is an essential concern in modern embedded systems [28] because of the small form factor and aggressive dynamic voltage scaling [16, 39].

Several techniques have been proposed to protect the embedded systems against soft errors. Hardware-based techniques based on modular [23] and information redundancy schemes [3] can protect processors by adding redundant hardware [24, 35]. Software-based protection techniques, such as instruction duplication [30, 38], have been proposed to avoid additional hardware costs. These schemes detect or correct soft errors by duplicating the instructions and validating the results between the original and duplicated instructions. However, these protection techniques are inappropriate for resource-constrained embedded systems because they are expensive and ineffective due to additional hardware modules or duplicated instructions.

Various efficient solutions have been proposed to mitigate protection overheads. Some techniques use unique characteristics of operations to provide low-cost protection. For example, algorithm-based fault tolerance techniques [7, 14] use the mathematical traits of matrix multiplication to generate efficient checksums. Some works claim to achieve low-cost protection for deep neural networks by restricting the magnitudes of activation functions [9, 13]. These techniques, however, are only applicable to specific operations. They also do not consider the vulnerability of their targets, which may lead to over-protection of resilient parts or under-protection of vulnerable parts of the system.

Selective or partial protection techniques are attractive alternatives, as the targets for protection are chosen flexibly. From the hardware perspective, system designers may choose to protect only the most vulnerable microarchitectural components. For example, the register file can be a candidate because soft errors in the register file can quickly propagate to other components [29]. The pipeline register is also considered vulnerable because it contains essential information between the pipeline stages [15]. From the software perspective, designers may choose to protect only the most vulnerable instructions. For example, Reis et al. [33] claimed that the program always returns the correct output if all memory write operations and control-flow instructions are executed correctly. However, these ideas are mainly based on heuristics, and the vulnerabilities of microarchitectural components and instructions against soft errors have not been comprehensively studied.

In this work, we present CFA, a comprehensive failure analysis framework to find the microarchitectural components and software instructions that are most vulnerable to soft errors [18]. Our failure analysis framework is based on the cycle-accurate system-level gem5 simulator [5]. It can inject soft-error-modeled faults into a microarchitectural component, log their impacts on system behaviors, and analyze system failures from both hardware and software perspectives. For the hardware perspective analysis, CFA injects faults into specific microarchitectural components to estimate the vulnerability of each component separately. For the software perspective analysis, CFA performs the root cause instruction analysis, a novel technique to find the software instructions responsible for system failure. CFA is publicly available at `https://github.com/dependablecomputinglab/CFA-framework`.

We perform vulnerability analysis from various perspectives on ten benchmarks from the MiBench [12] benchmark suite using our framework. From the hardware perspective, the pipeline register must be protected with the utmost priority because it contains essential metadata. From the software viewpoint, compare instructions showed the highest Architectural Vulnerability Factors. However, the results from individual perspectives are subject to change depending on the target application or the protection method. Therefore, choosing the hardware component or software instructions to protect must be decided comprehensively, taking multiple factors into account. To the best of our knowledge, this study is the first to highlight the various attributes

*Corresponding author
E-mail address: Yohan.Ko@yonsei.ac.kr

of instructions, hardware components, and the target system as a whole.

2. Related Works

2.1. Protection Techniques against Soft Errors

Several techniques have been proposed to protect the processors from soft errors. These techniques can be classified into two categories: hardware protection and software protection. Hardware solutions require the addition of redundant hardware [3, 23, 24, 35], which makes these solutions inflexible and expensive. The large area-overhead and high cost-overhead of full hardware protection techniques hinder them from being applied to resource-constrained embedded systems. For example, concrete shielding suggests shielding the system from external radiation by surrounding it with many feet of concrete [4, 20], which is simply inapplicable to such systems.

Software protection techniques do not apply any hardware modifications and instead rely on software methods such as instruction duplication. Oh et al. [30] proposed duplicating instructions and comparing the results between original and duplicated instructions to detect soft errors. While these techniques do not have the disadvantages of hardware protection techniques, software-based protection methods induce huge performance overheads since all instructions need to be duplicated [17]. This type of overhead is also intolerable for most embedded systems.

Because full protection is impractical for resource-constrained embedded systems, many techniques that reduce overheads in terms of area and performance have been proposed. One solution is to take advantage of the characteristics of specific operations to protect the operations with a much lower overhead than full replication. For example, algorithm-based fault tolerance techniques [7, 14] produce low-cost checksums for matrix multiplication operations. Another target is the activation function in deep neural networks. Simply limiting the range of these function outputs can effectively correct errors in the previous layers [9, 13]. Unfortunately, these techniques are inflexible as they can only protect their designated targets. They cannot be applied to systems that do not include the target operations.

Selective protection is a more flexible solution that offers high levels of error resilience with minimal overhead by protecting only the most vulnerable microarchitectural components and software instructions. Although selective protection is applicable to any system, the programmer must first know the most vulnerable parts of the system. Several arguments exist for the most vulnerable microarchitectural components. Naseer et al. [29] suggested that the register file could be the most vulnerable because errors in the register file can quickly and easily propagate to other components. Other candidates include the pipeline register, which holds essential information between pipeline stages [15], and the scoreboard, in which errors may destroy the data dependency between instructions [26].

There are various ideas on the most vulnerable software instruction types. In [33], store instructions were considered the most critical. The argument is that in memory-mapped I/O, a program always returns the correct output if all store instructions are executed correctly. Control-flow instructions are also considered vulnerable because an incorrect control-flow can execute incorrect store instructions or omit the execution of correct store instructions. On the other hand, Wei et al. [41] suggest that store instructions need not be protected. Most data stored by store instructions are overwritten before they are used, and therefore, protecting store instructions could result in a waste of resources. Instead, the authors claim that add instructions are the most vulnerable because they are frequently used to control loop index variables. Liu et al. [22] consider add instructions to be vulnerable for similar reasons.

2.2. Reliability Evaluation Techniques

Because soft errors are rare events in real-world environments, it is unrealistic to record the actual occurrence of soft errors to measure the reliability of a system. Therefore, reliability evaluation techniques involve numerous trials with manually induced soft errors and the analysis of the effect of those errors on the processor. One notable method for manually inducing errors is neutron beam testing [11]. Neutron beam testing exposes a processor to neutron-induced soft errors by placing a test processor in a cyclotron. Because this experimental environment is very similar to how soft errors occur, the results are highly accurate. However, beam testing experiments are costly to perform and require real hardware of the finished product [6]. Furthermore, beam testing experiments cannot show the reliability distribution among microarchitectural components in a processor because all components are simultaneously exposed to soft errors.

Fault injection campaigns have been presented as alternatives to expensive and complicated beam testing [1, 31, 32]. Fault injection campaigns intentionally inject faults into the system by reversing the value of a bit in the processor at a specific cycle during execution. These experiments can be performed even in the early design phase using a simulator. It is commonly considered that all bits in the system are equally likely to experience a soft error, and the bit and cycle are, therefore, chosen in a uniform random. With fault injection campaigns, it is possible to calculate the component-wise reliability. The vulnerability of each component can be estimated independently by injecting faults into the selected microarchitectural components.

Ideally, the reliability of a system can be measured by injecting soft errors into all the bits in each cycle. However, exhaustive injection experiments are impractical due to the tremendous number of trials needed, which is directly proportional to the number of bits in the program [10]. Therefore, most studies adopt the idea of statistical fault injection [21]. The idea is to perform a number of trials sufficient to make the results statistically significant but much less than the number required by exhaustive fault injection. For example, a statistical fault injection experiment may involve 10,000 injections. According to probability theory, a sample size of 10,000 is sufficient to achieve a 1% margin of error with a 95% confidence level, regardless of the population size.

Using the results from these experiments, one can calculate the system's Architectural Vulnerability Factor (AVF) [27]. This metric represents the probability of a fault to lead to a system failure and is defined as: $\frac{Total\ residency\ of\ ACE\ bits}{Total\ number\ of\ bits \times Total\ execution\ cycles}$, where ACE bits represent the bits that lead to failures when a fault is injected. If we consider failure cases of the injection trials as injections to ACE bits, this equation can be rewritten as: $\frac{Number\ of\ Failure\ Cases}{Total\ number\ of\ Injection\ Trials}$. Finally, multiplying the obtained AVF with the natural fault rate yields the FIT rate of the system.

3. Comprehensive Failure Analysis Framework

The most vulnerable hardware components or software instructions must be correctly identified to provide effective selective protection. However, existing selective protection methods rely primarily on heuristics when finding the vulnerable parts of a system. In addition, previous works approach this problem from only one perspective, and the vulnerabilities of the system against soft errors have not been studied comprehensively. Therefore, we developed CFA, a failure analysis framework based on the cycle-accurate gem5 simulator [5]. Our framework can inject soft error modeled faults into a microarchitectural component in an in-order CPU, log its impacts on system behaviors, and analyze system failures from both perspectives, as shown in Fig. 1.

The fault injection module enables the injection of single-bit soft

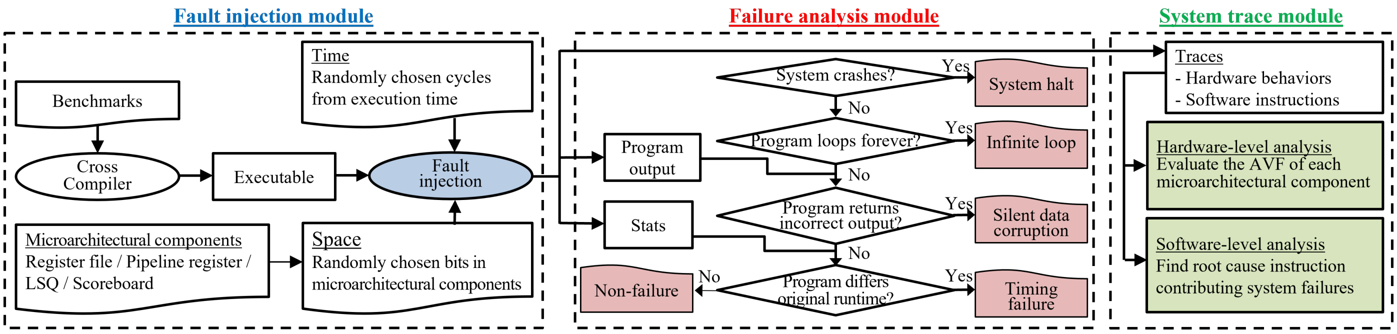

Fault injection module
Benchmarks
Time
Randomly chosen cycles from execution time
Cross Compiler
Executable
Fault injection
Microarchitectural components
Register file / Pipeline register / LSQ / Scoreboard
Space
Randomly chosen bits in microarchitectural components
Failure analysis module
Program output
Stats
System crashes?
Yes
System halt
No
Program loops forever?
Infinite loop
Program returns incorrect output?
Silent data corruption
Program differs original runtime?
Timing failure
Non-failure
System trace module
Traces
- Hardware behaviors
- Software instructions
Hardware-level analysis
Evaluate the AVF of each microarchitectural component
Software-level analysis
Find root cause instruction contributing system failures

Figure 1: Our gem5-based framework consists of a fault injection module to model soft errors, a failure analysis module to classify failure types, and a system trace module to analyze system failures from hardware and software perspectives.

errors to an in-order CPU by modifying the gem5 simulator. We found that too many injection cases in processors supporting out-of-order execution resulted in non-failures and chose to target the in-order CPU to reduce the number of non-failure cases. The in-order CPU also has the advantage of being easier to trace. The single-bit error is injected into a randomly selected bit at a randomly selected cycle within the execution time in one of the four main components (register file, pipeline register, load store queue (LSQ), and scoreboard). These components are representative of the other components, and similar targets have been chosen for fault injection in previous works [37, 38]. Notably, we did not inject errors into the cache. Due to the large size of the cache, errors occur much more frequently in caches than in other components, and therefore caches should be protected before any other components. In addition, previous works also excluded the cache and memory from their analyses, assuming that they are protected with parity or ECC [9, 33, 37]. With the fault injection module, CFA executes several iterations of a cross-compiled benchmark with the fault injections and returns simulation statistics, program outputs, and simulation traces composed of hardware-level microarchitectural behaviors and software-level instructions.

At the end of each injection trial, the failure analysis module classifies the type of system failure that occurred by comparing the program outputs and stats of the trial to those of the golden trial, the original trial with no injections. Based on the comparison results, the trial is classified as a non-failure case or one of four failure cases, as shown in Fig. 1. If the injected fault causes a system crash, such as a page table fault or segmentation fault during the simulation, it is categorized as *system halt* failure. If not, the runtime of the injected trial is checked to determine whether an infinite loop has occurred. We consider any trial executed for more than twice the original runtime as a case of an *infinite loop*. If the program terminates within twice the original runtime, but the output differs from the golden output, the trial is classified as a *silent data corruption* failure. If the program output is equal to that of the golden run, but the runtime differs from the original runtime, we classify it as a *timing* failure. Note that timing failures can be ignored if execution time is inconsequential, but they are considered for detailed analyses in this study. Finally, if the injected trial does not cause system failures and terminates within the same output and runtime as the original case, it is considered a non-failure case.

The system trace module analyzes each case of system failure to find the hardware component or software instruction responsible for the failure. Traces of hardware behaviors are used to evaluate the AVF of each microarchitectural component. The trace module analyzes all instructions that access the corrupted data for software instruction analysis. For example, suppose a fault is injected into the register file. In that case, instructions that read from the corrupted register file are considered vulnerable because they can cause erroneous system behaviors or propagate the fault to other components. When another instruction writes to the injected register file, it overwrites the faulty data and stops the error from impacting the system. Then, instructions following the write instructions become non-vulnerable.

Our framework can be used to perform extensive fault injection experiments over various benchmarks and calculate the vulnerability of the microarchitectural components and software instructions. The trace data and logs from injection trials can be used for further analyses. For example, if a trial resulted in one of the four failure cases, the information is used to pinpoint the root cause of the failure. Otherwise, it can help analyze how the induced fault was masked from the perspective of the system. Thus, our framework can act as a toolset for a comprehensive investigation of the impact of soft errors on the system from both hardware and software perspectives.

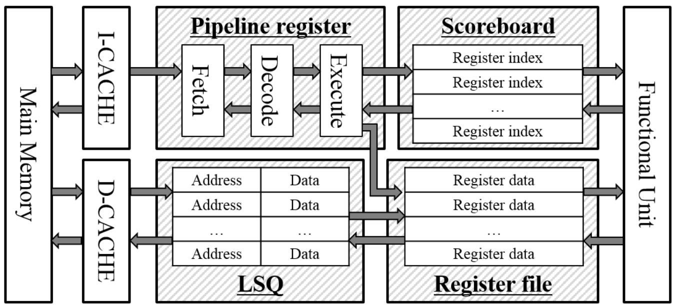
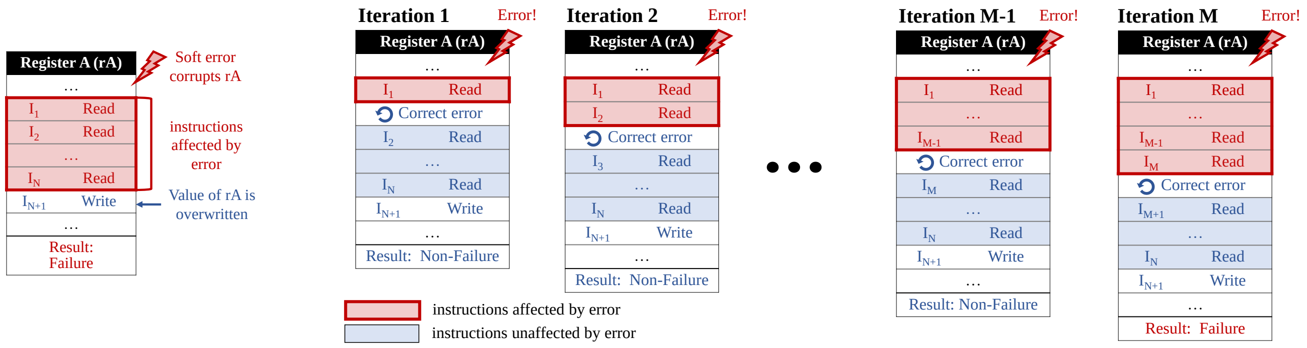


Figure 2: Microarchitectural components in a processor, where the shaded regions represent the fault injection targets.

3.1. Failures in Hardware Components

Injection-based failure analysis from the hardware perspective is relatively simple. Because we inject faults at the hardware level, we can directly map each fault to the injected microarchitectural component. Then, the injection trials for each microarchitectural component can be accumulated to calculate the component-wise AVFs. Trace data and logs of each trial are carefully analyzed to determine whether the injected fault becomes masked and determine the type of failure caused by the fault when it propagates to other components. We performed our analysis on the four main microarchitectural components shown in Fig. 2.

An error in the 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 error-injected bit after the cycle in which the error is injected. We use the instruction `load r1, r2`, which loads 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.

Figure 3: Two-level analysis to identify the root cause instructions when faults are injected into the register file

data in the memory address designated by register `r2` to register `r1`, to illustrate this process. First, consider the case in which the fault is injected into the register file. If the data in `r2` becomes corrupted, the processor may access the wrong data or, even worse, an invalid memory segment. If the data in `r1` becomes corrupted, the error is overwritten to a correct value and becomes masked. Our framework ceases to track the reads and writes on the corrupted bit in this case.

The process is similar when an error is injected into the load store queue (LSQ). 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 the memory data in the LSQ. Finally, `r1` is updated using the memory data in the LSQ. If the memory address is corrupted before the LSQ accesses the data cache, system failures, such as silent data corruption or system halt, can occur. Injection in the memory data after the LSQ loads the data from memory, but before `r1` is updated, would likely lead to silent data corruption. Other types of system failures may occur in rarer cases depending on where `r1` is used. For example, incorrect values of `r1` can induce timing failures if `r1` is used as the index of a loop or induce system halts if `r1` is used as the memory address in another memory instruction.

Now consider the case in which the error is injected into the pipeline register. During the pipeline stages, the pipeline register stores important information about the instructions, such as the opcode and indexes of the source and destination registers. If the injected fault corrupts the opcode, the `load` instruction could become any other type of instruction. This corruption may cause any of the four types of system 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`). The failure type due to this error depends on the program context, with silent data corruption being the most likely.

Finally, consider the case in which the error is injected into the scoreboard. Prior to the execution of the instruction `load r1, r2`, the destination register index (`r1`) is logged in the scoreboard. The register index stays on the scoreboard until the load instruction is completed to prevent any violations of data dependency. Other instructions using `r1` as a source register check the scoreboard and stall their execution. If the error causes the scoreboard to log `r0` instead of `r1`, the instructions using `r1` may not wait for the load instruction to complete. These instructions would then read the data of `r1` before the update. Using the incorrect value of `r1` could lead to various system failures. The error may also cause other instructions using `r0` to wait for the completion of the load instruction, which would most likely cause timing failures.

3.2. 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 behavior of soft errors in the real world. An additional step must be taken to determine the software instructions affected by the injected fault. In some instances, more than one instruction may access the injected bit, further complicating the issue. Previous works evaluating software vulnerabilities avoid this problem by injecting faults at the software level or restricting the injection sites to register files [22, 41]. To meet this issue, we propose the novel root cause instruction analysis, which maps each injected fault that causes a system failure to the root cause instruction, the single instruction responsible for the failure.

When faults are injected into microarchitectural components other than the register file, it is relatively simple to point out the root cause instruction. For example, if an error injected in the LSQ modifies a memory data or memory address, the instruction that inserts the value is labeled the root cause instruction. Likewise, if the injected fault alters an opcode or operands of an instruction, that instruction is the root cause instruction. The same applies to the scoreboard. In these components, the targeted bit can always be traced back to a single instruction, and the instruction is designated as the root cause instruction.

However, finding the root cause instruction is more challenging when errors are injected into register files. If a fault injected into a register is read by two or more instructions, and the system results in a failure, it is difficult to determine the single instruction responsible for the system failure. In this case, CFA follows a two-level analysis to identify the root cause instruction, as shown in Fig. 3. Assume that the data in a register file becomes corrupted, and the program results in a system failure. In the first-level analysis, CFA 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 without conducting the second-level analysis. In the other case where multiple instructions read the erroneous data, we conduct the second-level analysis to find the root cause instruction.

The core idea behind the second-level analysis is to isolate the effect of the error on one instruction at a time. The idea involves multiple iterations of the program with an error-correction function, which we 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. 3(a). In the first iteration, we correct the value of `rA` immediately after the first instruction (I_1) uses the value. In this way, the effect of the error is isolated to 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. For example, consider the M^{th} iteration in which the error is corrected after the M^{th} instruction reads the faulty value, and the iteration results in a system failure. Because the application did not fail when the first $M - 1$ instructions read the faulty value, our framework deduces that the M^{th} instruction is the root cause instruction. This process is illustrated in Fig. 3(b).

With the novel root cause instruction analysis, each injected fault that causes system failures can be traced to the instruction responsible for the failure. Then, the injected faults can be abstracted to the software instruction level without worrying about the details of the underlying hardware. This software perspective analysis can weigh the vulnerabilities of each instruction type in the benchmarks and explain which instructions should be protected when selective protection techniques are applied.

4. Failure Analysis and Protection Guidelines

We use our failure analysis framework to perform a comprehensive failure analysis on ten benchmarks from the MiBench [12] benchmark suite as a case study. We first perform analyses from each perspective to gain a fundamental understanding of the overall vulnerability of the hardware components and software instructions. This step incorporates the distribution of failure types to determine whether certain components or instructions are especially vulnerable to specific failures. Then, we examine how protection from one perspective affects vulnerability from the other perspective. We also explore how the system's resilience to certain failures affects the AVFs calculated from different perspectives. Finally, we present efficient selective protection guidelines for resource-constrained embedded systems based on the multi-perspective analysis results.

For ease of failure analysis, the scope of our experiments covers only single-bit errors. Not only are multi-bit errors much rarer than single-bit errors [19], but they also follow similar error propagation patterns as single-bit errors [34]. In addition, experiments performed by Chatzidimitriou et al. confirmed that error rates calculated from single-bit fault injection experiments are similar to those obtained from beam testing experiments [8]. Our analysis incorporates extensive fault injection campaigns targeting an in-order 32-bit ARM architecture simulated with the gem5 simulator [5]. The injection campaigns cover ten benchmarks (*basicmath*, *bitcount*, *crc*, *dijkstra*, *gsm*, *jpeg*, *matmul*, *qsort*, *stringsearch*, and *susan*) from the MiBench benchmark suite, with each injection trial consisting of a full execution of the application with one injected fault.

We performed a total of 400,000 fault injection campaigns, with 10,000 random faults per component for each benchmark. From the injection trials, we calculate the AVF as: $\frac{Number\ of\ Failure\ Cases}{Total\ number\ of\ Injection\ Trials}$. According to probability theory, 10,000 trials are sufficient to achieve a margin of error of 1% with a 95% confidence interval [21]. We executed additional simulations to confirm that this is true in our experiments. Incremental injection into the *bitcount* benchmark showed that the AVF varied within 1% after approximately 500 iterations. Thus, 10,000 injections are sufficient to analyze and validate system failures both theoretically and empirically.

Table 1
AVFs of each microarchitectural component

	Pipeline Register	Register File	LSQ	Scoreboard
AVF	44.05%	39.37%	31.19%	6.35%

We use the AVF metric to compare the relative vulnerabilities between components or instructions. Another popular metric is the FIT rate. The FIT rate also considers the size of the component and the fault rates and more accurately represents the system's reliability. However, using FIT rates has two main disadvantages in our domain. First, a larger hardware area results in higher protection costs as well as higher FIT rates. To apply the most efficient protection, we require a metric that represents the average vulnerability regardless of the size of the component. In addition, fault rates are usually calculated through complicated beam testing experiments. With the assumption that fault rates per bit are equal in all bits, the exact fault rates need not be known when comparing relative vulnerabilities between components of a system. For these reasons, we use the more simple AVF over FIT rates in our analysis.

4.1. Hardware-perspective Analysis Results

Table 1 summarizes the overall AVFs of the four microarchitectural components: pipeline register, register file, load store queue(LSQ), and scoreboard. The pipeline register is the most vulnerable microarchitectural component, with an AVF of 44%, closely followed by the register file with an AVF of 39%. The high AVF of the pipeline register is mainly because the pipeline register contains information crucial for correct execution, such as the operations and operands. Moreover, the pipeline register is filled with data to be used, making it experience less masking effects than other components. On the other hand, the scoreboard is the least vulnerable because faults in the scoreboard only affect the 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.

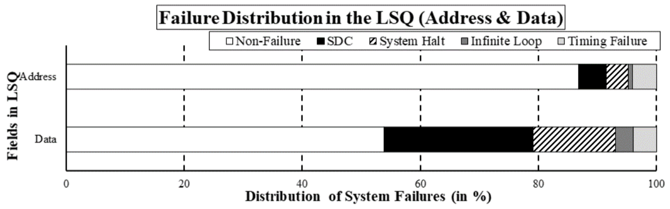


Figure 4: Detailed failure analysis of the LSQ. The chances of system failures are much higher when faults are injected in the memory data as opposed to the memory address.

Despite the general assumption that memory instructions are critical to correct execution, the AVF of the LSQ is not exceptionally high. Therefore, we examine in detail the LSQ in Fig. 4, which illustrates the classification of failures in the LSQ specifically. In this graph, the x-axis represents the distribution of system failures, including non-failures. The y-axis shows where the fault was injected: `Address` meaning that faults were injected into the memory address, and `Data` meaning that faults were injected into the memory data. Note that the rate of failures in the memory data is almost four times higher than that of the memory address (46% compared to 13%). 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

Table 2
AVFs of each type of instruction

	Store	Load	Arith	Logic	Cmp	Branch
AVF	38.23%	25.79%	29.03%	37.93%	39.45%	36.73%

data portion of the LSQ is non-vulnerable until the data is loaded from memory because the load will overwrite any errors that may have occurred. After the memory is accessed, 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. The higher failure rates of memory data imply that memory is accessed soon after the data and address are loaded to the LSQ, but that information stays in the LSQ for a longer time after the memory access. Therefore, in contrast to other memory components, the bits in the LSQ are only vulnerable for a limited amount of time. Then, the bits that become non-vulnerable shield the rest of the LSQ through spatial masking, reducing the AVF of the LSQ to approximately 30%.

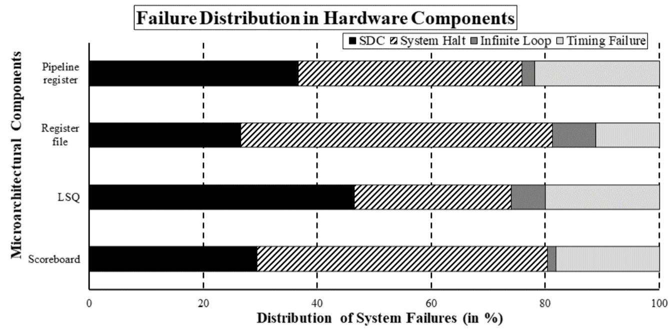


Figure 5: Component-wise distribution of system failures. Most components are particularly more vulnerable to a specific type of failure.

We also found that different hardware components are more vulnerable to specific types of failures. Fig. 5 shows a detailed failure analysis from the hardware perspective. The x-axis represents the distribution of system failures, and the y-axis lists the microarchitectural components. The graph shows that the failure cases of some components are concentrated in one type of failure. Nearly 47% of failures in the LSQ are silent data corruption failures, and system halt failures account for 55% and 51% of failures for the register file and scoreboard, respectively. These components can benefit from environments in which particular types of failures are tolerable. For example, if the target application is resilient against wrong outputs (e.g., multimedia applications), the LSQ becomes safer than the other components. In the case of the register file and scoreboard, the AVF can be effectively reduced by over 50% if the hardware allows recovery from system halt failures. However, the failure types are more evenly distributed in the case of the pipeline register. This implies that the pipeline register would remain the most vulnerable component across different environments, even when specific types of failures are no longer a concern.

4.2. Software-perspective Analysis Results

In our software-perspective analysis, we first categorized software instructions into six types: load, store, arithmetic, logical, compare, and branch. We then performed the root cause instruction analysis to pinpoint each failure to a single instruction. On average, the second step of the root cause instruction analysis was required for about 60% of the failure cases from the injections to register files. However, the computational overhead from the additional trials was only around 6% since around 80% of the cases required less than five additional trials. Note that the overhead could be further reduced by incorporating checkpointing ideas. Thus, the root cause instruction analysis is an effective and efficient technique to pinpoint the root cause instruction. With this novel technique, we could analyze system failures with different types of software instructions, as we did with hardware microarchitectural components.

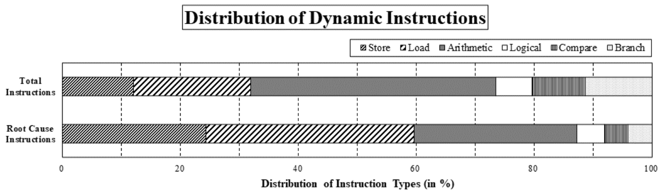


Figure 6: Distribution of dynamic instructions and root cause instructions. The metrics of store instructions show that protecting 12% of all instructions can reduce total failures by 24%.

We first compared the distribution of the dynamic instructions with the distribution of failure-inducing dynamic instructions, as shown in Fig. 6. The row on the top represents the occurrence rate of all dynamic instructions. For example, about 12% of all instructions are store instructions. The row on the bottom represents the distribution of root cause instructions; the graph shows that about 25% of all root cause instructions are store instructions. While store instructions take up only 12% of all dynamic instructions, about a quarter of all system failures were caused by store instructions. On the other hand, branch instructions have a similar dynamic occurrence rate to store instructions but are responsible for less than 5% of all failures. The disparity between these two statistics supports the case for efficient selective software instruction duplication. A similar conclusion was also noted by Liu et al. [22].

The AVFs of each type of root cause instruction are summarized in Table 2. It is interesting to note the disparity between the AVFs of memory instructions. Load instructions show relatively low AVFs of 25%, while store instructions have the second-highest AVF of 38%. We believe this is due to software-level masking opportunities. Load instructions are abundant at the beginning of functions, as they are used to initialize data. Therefore, faults in load instructions have plentiful opportunities to become masked before propagating to the final output. However, store instructions are frequently used at the end of functions or programs to return the final output. Faults in store operations directly affect the output, resulting in significantly higher AVFs for store instructions. This result is in tandem with previous works that suggested the protection of storing instructions.

We also considered the distribution of failure types for each type of instruction. In Fig. 7, the x-axis represents the distribution of system failures, and the y-axis represents the categorization of root cause instructions. Depending on the type of instruction that caused the failure, the most frequently occurring failure type is different. For instance, system halt failures are the most frequent in memory instructions, accounting for 45% and 58% of system failures when the type of root cause instruction is store and load, respectively. On the other hand, silent data corruption failures are most frequent in logical instructions (57%). This imbalance in failure types can be exploited for selective protection. For example, if wrong outputs are not a problem, system designers can prioritize the protection of instructions other than logical instructions.

The compare instructions have the highest proportion of timing

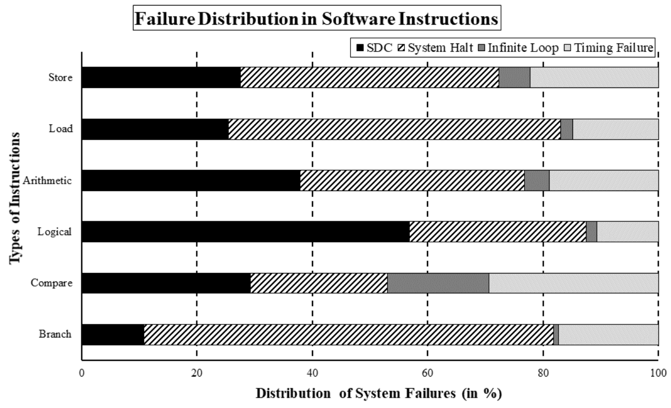


Figure 7: Distribution of system failures depending on the type of instruction. Most system failures caused by load and store instructions are system halt failures, while errors in arithmetic and logical instructions commonly lead to silent data corruption failures.

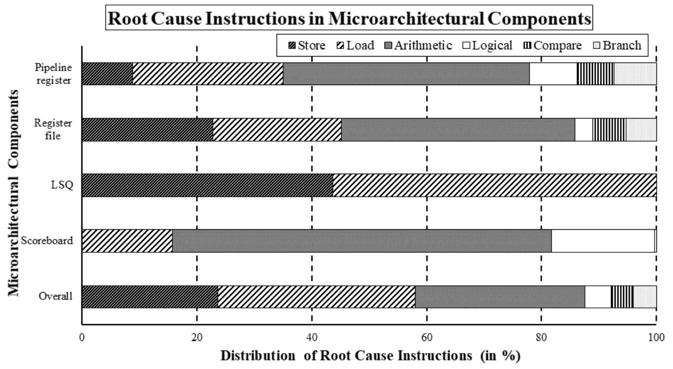


Figure 8: Distribution of failure-inducing instructions for each hardware component. Load instructions and arithmetic instructions account for the majority of failure cases.

Table 3
AVFs of instruction types before and after applying protection on the pipeline register

AVFs	Store	Load	Arith	Logic	Cmp	Branch
Before	38.23%	25.79%	29.03%	37.93%	39.45%	36.73%
After	33.34%	19.00%	15.79%	14.22%	16.99%	13.58%

Table 4
AVFs of microarchitectural components before and after applying protection on store instructions

AVFs	Pipeline Register	Register File	LSQ	Scoreboard
Before	44.05%	39.37%	31.19%	6.35%
After	39.89%	30.39%	17.59%	6.35%

failures. Nearly 30% of failures caused by compare instructions are timing failures, and if timing failures are considered non-failure cases, the AVF of control-flow instructions becomes less than 30%. The high proportion of timing failures is because compare instructions are often used to determine the direction of the branch instruction. This is in agreement with the study by Wang et al. [40], which states that even if incorrect branches are taken, programs can still terminate with correct outputs and slightly increased runtime. Another notable fact is that while LSQ showed a high vulnerability to silent data corruption failures (Fig. 5), memory instructions are generally more vulnerable to system halt failures. This implies that when faults are injected into hardware components other than the LSQ, there is a much higher chance of errors in the memory address being read. This causes memory instructions to access prohibited areas, leading to system halt failures. A deeper analysis from both the hardware and software perspectives is presented in the next section.

4.3. Multi-Perspective Comprehensive Analysis

This section draws intriguing conclusions by synchronously analyzing the vulnerabilities from the two perspectives. Fig. 8 summarizes the vulnerability of each component from a comprehensive viewpoint, showing the distribution of the types of root cause instructions for each microarchitectural component. The final row shows the overall distribution of instruction types that cause failures in the system. Note that the LSQ only contains memory instructions for apparent reasons. Store and compare instructions are excluded from the scoreboard because they do not have destination registers and do not access the scoreboard.

The pipeline register and register file in the first two rows have fairly similar distributions of instruction types. The proportion of logical instructions is notably different, covering only 3.1% of the register files but over 8.3% of the pipeline register. This difference is because of how easily logical instructions are masked in the register file. Assume an instruction calculates the `and` of two values, say `1100` and `0011`. Without any faults, the result is `0000` (`1100 & 0011`). However, even if a fault corrupts the value `1100` to `0100`, the result is still `0000` (`0100 & 0011`). In summary, if one source bit of logical `and` instruction is `0`, the resulting bit is `0` regardless of the other source bit. In the case of logical `or` instruction, the result is always `1` if one source bit is `1`. Because faults in the register files only corrupt the data, there is a high chance that the fault will be masked if the data is used in a logical operation. On the other hand, the pipeline register also contains the opcode or operands of logical instructions. When these metadata are corrupted in the pipeline register, the fault is not so easily masked and leads to failures. This supports the result from the hardware perspective, which suggests that the pipeline register is the most vulnerable hardware component.

We also explore how the protection applied from one perspective affects the analysis from another perspective. Since from Section 4.1 we concluded that the pipeline register is the most vulnerable, we observed the effects of protecting the pipeline register on the AVFs of software instructions. The results in Table 3 show that the AVFs of the instructions drop significantly when the pipeline register is protected. Because the branch and compare instructions are handled only by the pipeline register and register file, protecting one of the two components is sufficient to reduce these instructions' vulnerabilities drastically. On the other hand, the AVFs of the memory instructions remain relatively unaffected because significant portions of the memory instructions are handled in the LSQ. When the LSQ is protected instead, the AVFs of store and load instructions dropped to 15.39% and 12.10%, respectively.

Conversely, Table 4 shows how the protection of store instructions affects the AVFs of microarchitectural components. As expected, the AVF of the LSQ is affected the most, dropping from 31.19% to 17.59%. The AVF of the pipeline register remains the highest among the four microarchitectural components, implying that protecting the pipeline register and store instructions is a reasonable solution in terms of AVFs. Indeed, applying protection on the pipeline register and store instructions reduces the overall AVF of the system to 13.98%.

Table 5
The resulting AVFs and AVF reduction rates of protection techniques

	None	HW/SW	CFA	MIN	AVG
AVF	30.37%	13.98%	11.90%	27.77%	18.98%
Reduce Rate	0%	53.97%	60.83%	8.55%	37.50%

However, an exhaustive search of the 24 possible combinations of software and hardware protection techniques showed that protecting the LSQ and arithmetic instructions results in the lowest AVFs. While exploring protection techniques from the two perspectives in a sequential manner may reveal seemingly promising solutions, the two perspectives must be examined synchronously to find the optimal protection method. Another significant element to consider is the system's resilience to different types of failures, meaning that the system does not experience the type of failure owing to applied protection or natural characteristics. For example, if the system incorporates a watchdog to avoid system halt failures, the register file and branch instructions, which show the highest distribution of system halt failures from each category, could be left unprotected. Although applicable cases may not be common, these factors can dynamically reduce the overall AVF of the system without incurring any overhead.

4.4. Protection Guidelines using CFA

This section presents efficient selective protection guidelines for resource-constrained embedded systems in various environments based on our previous observations. In each environment, we assume that the system is resilient to a type of system failure. We then use our framework to find the best combination of software and hardware protection techniques and present its failure reduction rate compared to the system with no protection.

First, consider the case where the target architecture has no natural resilience to any system failure. Without any protection applied, the system has an overall AVF of 30.37%, as shown in Table 5. The hardware perspective analysis claims it is generally most efficient to protect the pipeline register. From the software perspective, compare instructions have the highest AVFs, but Table 3 shows that store instructions become the most vulnerable after the protection of the pipeline register. This combination of protecting the pipeline register and store instructions is represented as HW/SW in the table. The CFA column represents the combination we found using our framework, which protects the LSQ and arithmetic instructions. On the contrary, MIN represents the least optimal combination. Finally, AVG represents the average AVF of the 24 possible combinations. The final row labeled Reduce Rate measures the percentage decrease in AVF after applying the protection. From the table, it is clear that the multi-perspective failure analysis of our framework reaches a lower AVF (11.90%) compared to the method derived from single-perspective analyses (13.98%).

We also summarized the results for the cases where the target architecture is naturally resilient to a specific type of failure in Fig. 9. For example, some environments may have loose time constraints allowing timing failures to occur, while others may incorporate simple watchdog hardware to avoid system halt failures. We list four different metrics for each case, HW/SW, CFA, MIN, and AVG, as we did in Table 5. Again, we see that HW/SW fails to identify the optimal protection method. In the absence of SDC failures, HW/SW performs only slightly better than the average case, and when timing failures can be ignored, HW/SW performs worse than the average case. Contrarily, CFA finds the optimal protection method in every case, reducing the AVF by up to 82%. These results demonstrate the usefulness of our comprehensive failure analysis framework for selective protection of resource-constrained embedded systems.

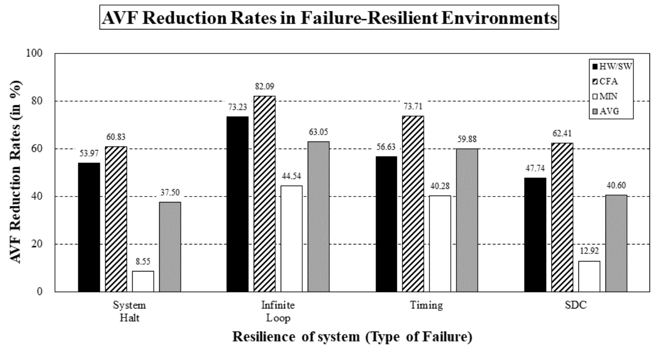


Figure 9: The AVF reduction rates in failure-resilient systems. CFA finds the optimal protection method for each case.

5. Conclusions

Soft errors are important reliability issues in the early design phase, but full protection against soft errors incurs severe hardware area and performance overheads. Efficient protection techniques have been proposed for resource-constrained embedded systems, but their robustness must be validated. Therefore, we present our comprehensive failure analysis framework, which can find the most vulnerable hardware component and software instruction through extensive fault injection campaigns. To the best of our knowledge, this framework is the first to consider how the protection of hardware components impacts the vulnerability of software instructions and vice versa to find the most efficient protection method for the system. We also use our framework to analyze the vulnerabilities of ten benchmarks from the MiBench benchmark suite. From each independent perspective, protecting the pipeline register (hardware) and store instructions (software) is considered the most effective. To maximize efficiency, however, the characteristics of the target application and the protection techniques from the two perspectives must be considered synchronously. With the multi-perspective analysis of CFA, we could reduce the AVF of a system by 82% (from 30.37% to 5.44%) by simply adding a watchdog and protecting the LSQ and arithmetic instructions.

6. 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), Institute of Information Communications Technology Planning Evaluation (IITP) grant funded by the Korean government (MSIT) (No. 2021-0-00155, Context and Activity Analysis-based Solution for Safe Childcare), and Samsung Electronics Co., Ltd(FOUNDRY-202108DD007F). We would like to thank Editage (www.editage.co.kr) for English language editing.

References

[1] Azimi, S., Du, B., Sterpone, L., 2017. Evaluation of transient errors in gpgpus for safety critical applications: An effective simulation-based fault injection environment. Journal of Systems Architecture 75, 95–106.

[2] Baumann, R., 2002. The impact of technology scaling on soft error rate performance and limits to the efficacy of error correction, in: IEDM.

[3] Baumann, R., 2005. Soft errors in advanced computer systems. IEEE Design & Test of Computers 22, 258–266.

[4] Baumann, R.C., 2001. Soft errors in advanced semiconductor devices-part i: the three radiation sources. IEEE Transactions on Device and Materials Reliability 1, 17–22.

[5] Binkert, N., Beckmann, B., Black, G., Reinhardt, S.K., Saidi, A., Basu, A., Hestness, J., Hower, D.R., Krishna, T., Sardashti, S., Sen, R., Sewell, K., Shoaib, M., Vaish, N., Hill, M.D., Wood, D.A., 2011. The gem5 simulator. SIGARCH Computer Architecture News 39, 1–7.

[6] Bodmann, P., Papadimitriou, G., Rech Junior, R.L., Gizopoulos, D., Rech, P., 2021. Soft error effects on arm microprocessors: Early estimations vs. chip measurements. IEEE Transactions on Computers , 1–1doi:10.1109/TC.2021.3128501.

[7] Bosilca, G., Delmas, R., Dongarra, J., Langou, J., 2009. Algorithm-based fault tolerance applied to high performance computing. Journal of Parallel and Distributed Computing 69, 410–416.

[8] Chatzidimitriou, A., Bodmann, P., Papadimitriou, G., Gizopoulos, D., Rech, P., 2019. Demystifying soft error assessment strategies on arm cpus: Microarchitectural fault injection vs. neutron beam experiments, in: 2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), IEEE. pp. 26–38.

[9] Chen, Z., Li, G., Pattabiraman, K., 2021. A low-cost fault corrector for deep neural networks through range restriction, in: 2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), IEEE. pp. 1–13.

[10] Chen, Z., Li, G., Pattabiraman, K., DeBardeleben, N., 2019. Binfi: an efficient fault injector for safety-critical machine learning systems, in: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 1–23.

[11] Dixit, A., Wood, A., 2011. The impact of new technology on soft error rates, in: IRPS, pp. 5B.4.1–5B.4.7.

[12] Guthaus, M.R., Ringenberg, J.S., Ernst, D., Austin, T.M., Mudge, T., Brown, R.B., 2001. MiBench: A free, commercially representative embedded benchmark suite, in: WWC, pp. 3–14.

[13] Hong, S., Frigo, P., Kaya, Y., Giuffrida, C., Dumitraș, T., 2019. Terminal brain damage: Exposing the graceless degradation in deep neural networks under hardware fault attacks, in: 28th USENIX Security Symposium (USENIX Security 19), pp. 497–514.

[14] Huang, K.H., Abraham, J.A., 1984. Algorithm-based fault tolerance for matrix operations. IEEE transactions on computers 100, 518–528.

[15] Jeyapaul, R., Flores, R., Avila, A., Shrivastava, A., 2016. Systematic methodology for the quantitative analysis of pipeline-register reliability. IEEE Transactions on Very Large Scale Integration (VLSI) Systems PP, 1–9.

[16] Kastensmidt, F.L., Tonfat, J., Both, T., Rech, P., Wirth, G., Reis, R., Bruguier, F., Benoit, P., Torres, L., Frost, C., 2014. Voltage scaling and aging effects on soft error rate in SRAM-based FPGAs. Elsevier Microelectronics Reliability 54, 2344–2348.

[17] Ko, Y., Kang, J., Lee, J., Kim, Y., Kim, J., So, H., Lee, K., Paek, Y., 2016. Software-based selective validation techniques for robust cgras against soft errors. ACM Transactions on Embedded Computing Systems 15, 20:1–20:26.

[18] Ko, Y., So, H., Jung, J., Lee, K., Shrivastava, A., 2021. Comprehensive failure analysis against soft errors from hardware and software perspectives, in: 2021 IEEE 39th International Conference on Computer Design (ICCD), IEEE.

[19] Lee, K., Shrivastava, A., Kim, M., Dutt, N., Venkatasubramanian, N., 2008. Mitigating the impact of hardware defects on multimedia applications: A cross-layer approach, in: MM, pp. 319–328.

[20] Lesea, A., Drimer, S., Fabula, J.J., Carmichael, C., Alfke, P., 2005. The rosetta experiment: atmospheric soft error rate testing in differing technology FPGAs. IEEE Transactions on Device and Materials Reliability 5, 317–328.

[21] Leveugle, R., Calvez, A., Maistri, P., Vanhauwaert, P., 2009. Statistical fault injection: Quantified error and confidence, in: DATE, pp. 502–506.

[22] Liu, Z., Liu, Y., Chen, Z., Guo, G., Wang, H., 2021. Analyzing and increasing soft error resilience of deep neural networks on arm processors. Microelectronics Reliability 124, 114331. URL: https://www.sciencedirect.com/science/article/pii/S0026271421002973, doi:https://doi.org/10.1016/j.microrel.2021.114331.

[23] Lyons, R.E., Vanderkulk, W., 1962. The use of triple-modular redundancy to improve computer reliability. IBM Journal of Research and Development 6, 200–209.

[24] Mallavarapu, P., Upadhyay, H.N., Rajkumar, G., Elamaran, V., 2017. Fault-tolerant digital filters on fpga using hardware redundancy techniques, in: 2017 International conference of Electronics, Communication and Aerospace Technology (ICECA), IEEE. pp. 256–259.

[25] Mittal, S., Inukonda, M.S., 2018. A survey of techniques for improving error-resilience of dram. Journal of Systems Architecture 91, 11–40.

[26] Monferrer, P.C., Vera, X., Abella, J., Casado, J.C., 2007. Mechanism for soft error detection and recovery in issue queues. US Patent App. 11/999,787.

[27] Mukherjee, S.S., Weaver, C., Emer, J., Reinhardt, S.K., Austin, T., 2003. A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor, in: Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003. MICRO-36., IEEE. pp. 29–40.

[28] Narayanan, V., Xie, Y., 2006. Reliability concerns in embedded system designs. Computer 39, 118–120.

[29] Naseer, R., Bhatti, R.Z., Draper, J., 2006. Analysis of soft error mitigation techniques for register files in IBM Cu-08 90nm technology, in: MWSCAS, pp. 515–519.

[30] Oh, N., Shirvani, P.P., McCluskey, E.J., 2002. Error detection by duplicated instructions in super-scalar processors. IEEE Transactions on Reliability 51, 63–75.

[31] Parasyris, K., Tziantzoulis, G., Antonopoulos, C.D., Bellas, N., 2014. GemFI: A fault injection tool for studying the behavior of applications on unreliable substrates, in: DSN, pp. 622–629.

[32] Reagen, B., Gupta, U., Pentecost, L., Whatmough, P., Lee, S.K., Mulholland, N., Brooks, D., Wei, G.Y., 2018. Ares: A framework for quantifying the resilience of deep neural networks, in: 2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC), IEEE. pp. 1–6.

[33] Reis, G.A., Chang, J., Vachharajani, N., Rangan, R., August, D.I., 2005. SWIFT: Software implemented fault tolerance, in: CGO, pp. 243–254.

[34] Sangchoolie, B., Pattabiraman, K., Karlsson, J., 2017. One bit is (not) enough: An empirical study of the impact of single and multiple bit-flip errors, in: 2017 47th annual IEEE/IFIP international conference on dependable systems and networks (DSN), IEEE. pp. 97–108.

[35] Schweizer, T., Schlicker, P., Eisenhardt, S., Kuhn, T., Rosenstiel, W., 2011. Low-cost tmr for fault-tolerance on coarse-grained reconfigurable architectures, in: 2011 International Conference on Reconfigurable Computing and FPGAs, IEEE. pp. 135–140.

[36] Seifert, N., Gill, B., Jahinuzzaman, S., Basile, J., Ambrose, V., Shi, Q., Allmon, R., Bramnik, A., 2012. Soft error susceptibilities of 22 nm tri-gate devices. IEEE Transactions on Nuclear Science 59, 2666–2673.

[37] So, H., Didehban, M., Ko, Y., Shrivastava, A., Lee, K., 2018. Expert: Effective and flexible error protection by redundant multithreading, in: 2018 Design, Automation & Test in Europe Conference & Exhibition (DATE), IEEE. pp. 533–538.

[38] So, H., Didehban, M., Shrivastava, A., Lee, K., 2019. A software-level redundant multithreading for soft/hard error detection and recovery, in: 2019 Design, Automation & Test in Europe Conference & Exhibition (DATE), IEEE. pp. 1559–1562.

[39] Wang, L., Skadron, K., 2013. Implications of the power wall: Dim cores and reconfigurable logic. IEEE Micro 33, 40–48.

[40] Wang, N., Fertig, M., Patel, S., 2003. Y-branches: when you come to a fork in the road, take it, in: PACT, pp. 56–66.

[41] Wei, X., Yue, H., Gao, S., Li, L., Zhang, R., Tan, J., 2020. G-seap: Analyzing and characterizing soft-error aware approximation in gpgpus. Future Generation Computer Systems 109, 262–274.

Jinhyo Jung is a PhD student in Dependable Computing Lab (DClab) at Yonsei University (http://http://dclab.yonsei.ac.kr/). He received Bachelor's degree in Computer science from Yonsei University, and currently he is in integrated PhD Course at the same university. His research interests include reliability issues such as comprehensive vulnerability estimation of computer architecture and protection schemes against soft and hard errors for deep neural networks.

Yohan Ko is currently an assistant professor in the Division of Software at Yonsei University, Wonju, Gangwon, Republic of Korea. He received his Ph.D. at the Department of Computer Science and Engineering at Yonsei University, Seoul, Republic of Korea.

Hwisoo So is a PhD student in Dependable Computing Lab (DClab) at Yonsei University (http://http://dclab.yonsei.ac.kr/). He received Bachelor's degree in Computer science from Yonsei University, and currently he is in integrated PhD Course at the same university. He is participating for Global PhD fellowship in National Research Foundation of Korea. His research interests include reliability issues such as comprehensive vulnerability estimation of computer architecture and hardware/software based protection schemes against soft and hard errors based on redundancy.

Kyoungwoo Lee is an associate professor in the department of computer science and engineering at Yonsei University, Seoul, South Korea.He received B.S. and M.S. degrees in computer science from Yonsei University in 1995 and 1997, respectively, and Ph.D. degree in information and computer science at the University of California at Irvine in 2008. His research is in the area of embedded systems, with a specific focus on cross-layer design and optimization for error-aware and energy-efficient embedded systems.

Aviral Shrivastava is an Associate Professor in the School of Computing Informatics and Decision Systems Engineering at the Arizona State University, where he has established and heads the Make Programming Simple (MPS) Lab (http://aviral.lab.asu.edu/).

He received his Ph.D. and Masters in Information and Computer Science from the University of California, Irvine, and bachelors in Computer Science and Engineering from Indian Institute of Technology, Delhi. Prof. Shrivastava's research lies in the broad area of Software for Embedded and Cyber-Physical Systems. He is currently serving as associate editor for ACM Transactions Embedded Computing Systems (ACM TECS), IEEE Transactions on MultiScale Computing (IEEE TMSC), IEEE Transactions on Computer-Aided Design (IEEE TCAD), and Springer International Journal on Parallel Processing (Springer IJPP), and Springer Design Automation for Embedded Systems (Springer DAEM). He is currently the program chair of CODES+ISSS 2017, one of the top conferences in embedded systems.