electronics

MDPI

Article

Characterizing System-Level Masking Effects against Soft Errors

Yohan Ko

Division of Software, Yonsei University, Wonju 26493, Gangwon-do, Korea; yohan.ko@yonsei.ac.kr

Abstract: From early design phases to final release, the reliability of modern embedded systems against soft errors should be carefully considered. Several schemes have been proposed to protect embedded systems against soft errors, but they are neither always functional nor robust, even with expensive overhead in terms of hardware area, performance, and power consumption. Thus, system designers need to estimate reliability quantitatively to apply appropriate protection techniques for resource-constrained embedded systems. Vulnerability modeling based on lifetime analysis is one of the most efficient ways to quantify system reliability against soft errors. However, lifetime analysis can be inaccurate, mainly because it fails to comprehensively capture several system-level masking effects. This study analyzes and characterizes microarchitecture-level and software-level masking effects by developing an automated framework with exhaustive fault injections (i.e., soft errors) based on a cycle-accurate gem5 simulator. We injected faults into a register file because errors in the register file can easily be propagated to other components in a processor. We found that only 5% of injected faults can cause system failures on an average over benchmarks, mainly from the MiBench suite. Further analyses showed that 71% of soft errors are overwritten by write operations before being used, and the CPU does not use 20% of soft errors at all after fault injections. The remainder are also masked by several software-level masking effects, such as dynamically dead instructions, compare and logical instructions that do not change the result, and incorrect control flows that do not affect program outputs.

Keywords: soft error; reliability; masking

check for updates

Citation: Ko, Y. Characterizing System-Level Masking Effects against Soft Errors. *Electronics* **2021**, *10*, 2286. https://doi.org/10.3390/electronics10182286

Academic Editor: Luis Gomes

Received: 25 August 2021
Accepted: 16 September 2021
Published: 17 September 2021

1. Introduction

Reliability against soft errors is becoming one of the most crucial design concerns in modern embedded systems [1]. Soft errors are transient faults in semiconductor devices caused by external radiation, such as alpha particles, thermal neutrons, cross-talk, and cosmic rays [2]. The soft error rate is exponentially increasing, mainly because of the reduction in chip size and supply voltage as technology scales [3]. Several techniques have been presented in various layers to protect embedded systems against soft errors. Because most protection techniques are based on redundancy methods, they should be more complicated than unprotected architectures, that is, larger hardware areas, longer execution times, or both. However, soft error protection techniques can be expensive or ineffective and sometimes fail to protect systemss [4]. Thus, we need to quantify the system reliability against soft errors in a quantitative and accurate manner to apply appropriate protection for resource-constrained or hard real-time embedded systems.

To quantify system reliability against soft errors, accelerated radiation beam testing [5] and exhaustive fault injection campaigns [6] have been used. Beam testing and fault injections are the two most accurate methods for representing system reliability against soft errors. However, they are not only expensive to perform but also challenging to set up correctly. Vulnerability modeling is an alternative method to evaluate reliability against soft errors [7] efficiently. Vulnerability estimation tools based on cycle-accurate simulators [8,9] are much faster than beam testing or fault injection campaigns because they can return vulnerability using only a single simulation. However, vulnerability estimation can be inaccurate, mainly due to several existing masking effects, because it only considers microarchitecture-level behaviors and ignores software-level masking effects. To the best of our knowledge, there have been no research works to analyze masking effects at the system level.

In this study, we developed and implemented a system-level framework designed to characterize soft error masking effects and analyze their distributions based on a cycle-accurate gem5 simulator [10] by considering both microarchitecture-level and software-level masking effects. We injected faults into the register file in an out-of-order processor for comprehensive system-level masking effects. Because errors in register files can be quickly and frequently propagated to other components [11], they can represent the fault injection into all the microarchitectural components in the out-of-order processor [12]. Furthermore, the register file is among the most sensitive components in a processor. The register file is essential to modern processors, and it usually holds data for a long time before being used.

We injected faults (soft errors) into randomly chosen bits in the register file at a randomly selected time during the execution time. Then, our framework determined whether an injected fault induces system failures. Existing fault injection approaches [13,14] focus on the failure rate, and they can return the number of injected faults that are eventually masked. However, our framework provides not only the failure rate but also the distribution of the masking effects from fault injection campaigns. Furthermore, if an inject fault does not cause any system failures (i.e., an error is masked), our framework analyzes the masking effect at the system level: (i) microarchitecture level and (ii) software level.

First, our framework analyzes the microarchitecture-level masking effects based on the microarchitectural behaviors of the components. For example, if committed instructions do not read the corrupted data due to soft errors in the register file, the corrupted data on the register file do not cause system failure or result in incorrect output. Our framework needs to trace microarchitecture-level behaviors (such as read and write operations) and analyze their masking effects to decide whether committed instructions read corrupted data. For instance, it is not vulnerable if write operations overwrite corrupted data before they are read or used for program execution.

Second, we also need to analyze masking effects at the software level when corrupted data is read by committed instructions but correct output. For instance, a compare instruction compares two source operands and sets the flag bits accordingly, but soft errors on one source operand may not change flag bits or affect systems at all (e.g., CMP *r*0, 0x10 and *r*0 has been erroneous from 0x00 to 0x01). Because the compare instruction reads the corrupted data in *r*0, the microarchitecture-level analysis determines this behavior as vulnerable. However, it does not affect the program, and it is not vulnerable. Thus, it is necessary to characterize and analyze them at the software level and accurately estimate the system-level susceptibility to soft errors.

Interestingly, our experimental results revealed that only 5% of the injected faults on the register file in an out-of-order processor cause system failures, such as segmentation faults, infinite loops, or incorrect outputs from several benchmarks, but primarily from the MiBench benchmark suite [15]. Thus, we need to analyze system-level masking effects to understand why 95% of the injected faults do not cause system failures. First, our system-level analysis framework traces microarchitectural behaviors, such as reads and writes and instruction commitments, to answer whether committed instructions read corrupted data. If not (e.g., corrupted data is overwritten or program execution does not use corrupted data), they are not vulnerable and do not affect the program. Our framework has shown that, interestingly, 91% of fault injections are masked on average over benchmarks because committed instructions do not read them.

If program execution reads corrupted data on the register file, we need to analyze the masking effects at the software level, such as assembly/instruction level. Interestingly, our experimental results show that almost 40% of the injected faults do not cause any system failures, even though committed instructions read them. Therefore, our framework compared outputs and execution traces (e.g., instructions, register data, and memory data) between the original execution and corrupted execution with a fault injection campaign to analyze the masking effects at the software level. If their program outputs are identical, execution traces are evaluated to investigate the masking effects at the software level. Furthermore, we traced the propagation of soft errors to other registers and memory components, and our framework analyzes the masking effects from propagated errors.

Our experimental results on several benchmarks show several interesting results based on a software-level masking effect analysis. First off, more than half of the software-level masking effects come from dynamically dead instructions, whose results are no longer used nor affect system behaviors. Second, when injected faults were masked at the software level, approximately 17% and 12% of soft errors induced corrupted operands in comparison and logical instructions, respectively. However, the results of these instructions were identical to those of the original execution without soft errors. Finally, approximately 16% of software-level masking effects take the incorrect branch, but they can still result in the correct output with with slightly affected execution time (at most 2%).

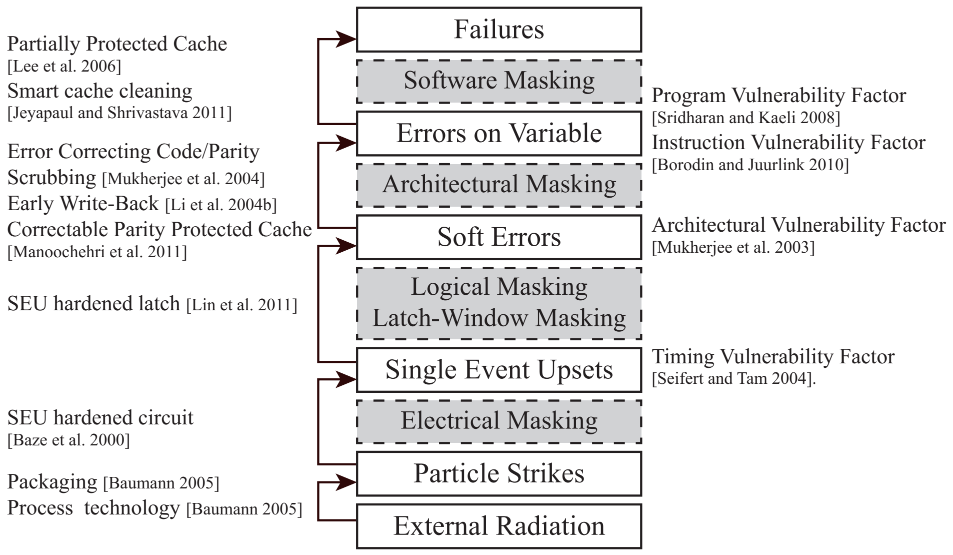
The remainder of this paper is structured as follows. In Section 2, we introduce related research on masking effects and reliability quantification at various layers. Next, the theoretical background of system-level masking effects is described in Section 3, and we explain our masking effect analysis framework in Section 4. Then, our experimental observations based on our framework are described in Section 5. Finally, Section 6 concludes this paper and indicates directions for future research.

2. Related Works

Soft errors are becoming more crucial in the reliability of semiconductor devices with technological advances, although soft errors are transient bit flips and nondestructive. For instance, the flagship server of Sun Microsystems crashed because of cosmic ray strikes on an L2 cache, and Sun suffered significant financial losses [16]. Furthermore, a soft error can cause failures in safety-critical systems, such as automotive, aerospace, public transportation, and healthcare systems. For instance, an embedded research group proposed that sudden unintended acceleration of vehicles can occur because of single-bit flips on engine control units [17]. Thus, several protection schemes have been proposed to protect embedded processors, but they can incur overhead in terms of performance, energy consumption, and hardware area. Therefore, we need to quantify the reliability of embedded systems accurately, and vulnerability modeling is an effective method. However, vulnerability estimation based on lifetime analysis should consider several masking effects at various layers for accurate modeling.

2.1. Masking Effects at Different Layers

Soft errors are transient faults in semiconductor devices caused by external radiation, such as energetic particles and cosmic rays [2]. Energetic particles strike the sensitive region of the transistors, causing a single-event upset (SEU). However, not all SEUs result in system failures owing to several masking effects, as shown in Figure 1. At the circuit level, SEU is latched and activated as a soft error unless an upset is masked by the latching-window masking (upset does not reach the latching-window time), electrical masking (upset is not strong enough to reach the latching element), and logical masking (upset on the input of a gate does not affect its output). At the microarchitectural level, latched soft errors are propagated to the error in the microarchitectural components unless they are masked by architectural masking. A soft error occurs on a microarchitectural component, but they are overwritten or not used for execution. This erroneous variable causes a failure at the software level, such as incorrect output or system crash, unless software-level masking effects hide it. Examples of software-level masking effects include dynamic dead code (for example, a code is executed, but it does not affect the program output) and data value masking (for example, if variable A is multiplied by zero, then A does not matter). It is challenging to capture and reflect all of these software-masking effects on vulnerability estimation at the microarchitectural level.



Failures
Software Masking
Errors on Variable
Architectural Masking
Soft Errors
Logical Masking
Latch-Window Masking
Single Event Upsets
Electrical Masking
Particle Strikes
External Radiation
Partially Protected Cache [Lee et al. 2006]
Smart cache cleaning [Jeyapaul and Shrivastava 2011]
Error Correcting Code/Parity Scrubbing [Mukherjee et al. 2004]
Early Write-Back [Li et al. 2004b]
Correctable Parity Protected Cache [Manoochehri et al. 2011]
SEU hardened latch [Lin et al. 2011]
SEU hardened circuit [Baze et al. 2000]
Packaging [Baumann 2005]
Process technology [Baumann 2005]
Program Vulnerability Factor [Sridharan and Kaeli 2008]
Instruction Vulnerability Factor [Borodin and Juurlink 2010]
Architectural Vulnerability Factor [Mukherjee et al. 2003]
Timing Vulnerability Factor [Seifert and Tam 2004].

Figure 1. Masking effects, protection techniques, and vulnerability estimation from circuit-level to software-level.

Several techniques have been proposed at the chip level to protect hardware against soft errors. To reduce the soft error rate, Baumann [18] presented packaging and process technologies that shield hardware from external radiation or modify the process. Polyimide thin films or concrete are proven shields against soft errors caused by alpha particles and cosmic rays. However, shielding techniques cannot protect the chip entirely against external radiation. For example, a neutron can pass through five feet of concrete. Hardware shielding techniques also incur significant overhead in terms of the hardware area and cost. Electrical masking effects can mask particle strikes because of the attenuated pulse caused by the electrical resistance in the circuit. In Reference [19], actively biased and isolated transistors in the circuit were proposed to reduce transient faults. Circuit-level protection can be applied to various hardware architectures, such as SRAM and combinational logic. However, they require additional hardware costs for modifying circuit designs.

SEUs can also be masked by logical and latch-window (or temporal) masking effects at the circuit level. Krishnaswamy et al. [20] proposed circuit modification for efficient redundant techniques that increase the logical masking effects. SEUs can be masked by latching-window masking effects when soft errors are not propagated owing to the required setup and hold time. Lin et al. [21] proposed an SEU-hardened latch design to improve hardware reliability by extending the setup time at the clock edge. However, circuit-level protection techniques induce massive overheads in area and performance due to hardware modifications at the hardware level.

Soft errors can be masked by architectural masking, for example, errors in microarchitectural components in a processor that are overwritten or not used in the execution at the microarchitectural level. Li et al. [22] proposed an early write-back policy to improve the reliability of cache memory without the additional area cost. The early write-back policy combines the performance efficiency of write-back with the high reliability of the write-through policy by exploiting the least recently used (LRU) algorithm or dead-time based approaches. In addition, Manoochehri et al. [23] proposed a correctable parity-protected cache to correct soft errors using an error detection code instead of more expensive error-correcting code. They corrected soft errors, including spatial multi-bit errors in the dirty state using multi-dimensional parity code without severe overheads in terms of performance and area.

All the errors in program variables do not induce system failures owing to the software masking effects. For instance, errors in multimedia data in multimedia applications can degrade the quality of service, but they do not induce system failures. Partially detected cache [24] has improved hardware reliability with comparable performance overheads. They protect only failure-critical data, such as control variables, based on data profiling; however, they do not protect multimedia data. Smart cache cleaning [25] detects specific cache blocks at specific periods by applying the hardware-software hybrid methodology. We can protect data efficiently using software-based or hybrid-based protective schemes at the software level, but the decision of importance in data or instructions is a very complex task.

2.2. Reliability Quantification at Different Layers

Seifert et al. presented the timing vulnerability factor (TVF) for a circuit environment where sequential elements are typically placed [26]. The particle strikes do not appear as soft errors on architectures because of electrical, logical, and latch-window masking effects. For example, assume that the latch accepts the data during half of the lifetime. The corrupted data at the circuit level will be masked by up to 50%. Such a circuit-level vulnerability factor is related to the raw-device fault rate, so it is challenging to consider the characteristics of each microarchitectural component. In addition, TVF does not consider the architectural masking effects because it is unaware of the behaviors of microarchitectural components. Thus, it is too conservative in expressing the vulnerability of each microarchitectural component.

Mukherjee et al. [7] proposed the architectural vulnerability factor (AVF) to estimate the reliability of each microarchitectural component accurately. AVF means the probability that a state change (soft errors or transient bit flips) in the device leads to an architecturally visible error. AVF analysis traces the architecturally correct execution (ACE) bits, which induce system failures if they change, and they estimate the resident time of ACE bits in microarchitectural components. Thus, its estimation was faster than that of the register-transfer level analysis models, but it was still inaccurate to analyze the reliability. Moreover, it is too conservative in estimating the reliability of microarchitectural components because it defines unconfirmed data that affect the final output. Further, architectural vulnerability estimation excludes the software-level masking effects to overestimate the reliability of each microarchitectural component.

Sridharan et al. proposed a software-level vulnerability, program vulnerability factor (PVF) [27]. PVF can be quickly estimated from the assembly code compared to the conventional vulnerability estimation, and it can be a good predictor for determining the AVF of hardware components. The instruction vulnerability factor (IVF) [28] evaluates the number of faults in all the instructions that affect the final program output. However, PVF and IVF are unaware of the hardware architectures because they are only based on software or assembly level analysis.

2.3. Fault Injection Approaches

Reliability quantification based on lifetime analysis can be inaccurate owing to system-level masking effects; thus, several fault injection approaches have been presented for accurate reliability qualification, as shown in Table 1. They injected faults into microarchitectural components in a processor to determine whether the injected faults caused system failures. The failure rate is defined as the number of faults that induce system failures divided by the number of fault injections. For instance, if 300 errors caused system failures of 1000 fault injections, the failure rate was 30% (=300/1000).

Table 1. Various fault injection approaches.

Type	Pros	Cons
Exhaustive fault injection [29]	Provide high accuracy	Need many numbers of fault injections
Pseudorandom fault injection [30]	Provide moderate accuracy	Still need lots of numbers of fault injections
Statistical fault injection [31]	Need a small number of fault injections	Provide low accuracy

Because exhaustive fault injection campaigns are time-consuming tasks, a pseudorandom and statistical fault injection campaign has been proposed to reduce the number of fault injections [30,31], as shown in Table 1. We assume that we need to extract the failure rate of the register file in a processor. The size of the register file in this processor is 500 bits, and the total execution time is 400 cycles. In this case, we must inject 200,000 (=500 $\times$ 400) faults if all possible instances are to be performed [29]. However, it is almost impossible to consider all possible cases from the temporal and spatial domains. Instead, randomly selected fault injections can represent exhaustive fault injections based on probabilistic theory.

Fault injection schemes can accurately provide the failure rate, but they do not explain why the injected faults are masked. For instance, if the failure rate of a component is 30%, approximately 70% of the injected faults are masked. However, existing fault injection frameworks [13,14] only trace whether injected faults induce system failures. In this study, we characterized system-level masking effects to analyze not only how many faults are masked but also why faults are masked.

3. Classification When Soft Errors Occur

Figure 2 shows the classification model when a soft error corrupts the data in the microstructural components. Existing fault injection schemes [13,14] only consider whether soft errors induce failures. On the other hand, our framework analyzes system-level masking effects both at the microarchitecture-level and at the software level because not all the faults in microarchitectural components cause system failures. Microarchitecture-level analysis determines whether corrupted data is used or read for program execution. The software-level analysis traces software-level behaviors to analyze masking effects when corrupted data is read but still results in the correct output.

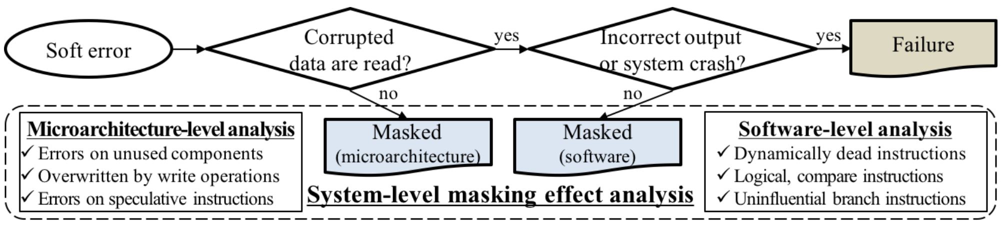

Soft error
Corrupted data are read?
yes
no
Incorrect output or system crash?
yes
no
Failure
Microarchitecture-level analysis
✓ Errors on unused components
✓ Overwritten by write operations
✓ Errors on speculative instructions
Masked (microarchitecture)
Masked (software)
System-level masking effect analysis
Software-level analysis
✓ Dynamically dead instructions
✓ Logical, compare instructions
✓ Uninfluential branch instructions

Figure 2. Classification when soft error occurs in microarchitectural components.

As shown in Figure 2, corrupted data in microarchitectural components can be masked at the microarchitectural level if committed instructions do not read corrupted data. Furthermore, soft errors do not affect the program execution when corrupted data is not used, overwritten, or used by mispredicted instructions (i.e., uncommitted/squashed instructions) in the out-of-order execution. These masking effects can be analyzed by tracing microarchitectural behaviors, such as reads and writes of each microarchitectural component and instruction commitment.

If committed instructions read corrupted data, it is necessary to check whether they cause a system crash. If the system halts (i.e., segmentation faults or system crashes) or program loops forever (i.e., infinite loop), we define these instances as system failures. If the program returns output without system failures, we must compare the output with the original one. If their outputs are different, they are also defined as failures (i.e., silent data corruption). Even though committed instructions read corrupted data, they can still produce the correct output because of several software-level masking effects.

Soft errors do not induce failures if dynamically dead instructions read corrupted data caused by soft errors. Dynamically dead instruction is an instruction whose result is no longer used or affects system behaviors [32]. Statically dead instructions can be excluded from the assembly code at compile-time, but dynamically dead instructions cannot be determined before execution. For instance, instructions in the loop can be executed many times, and only a part of the instructions can be dynamically dead based on the context. Thus, we need to trace whether other instructions use corrupted results of instructions by analyzing the data flow of a program.

In Figure 3, instruction A (STORE $r1$, 0x100) stores data contained in $r1$ into the memory 0x100. If data contained in $r1$ is changed to 0x11 from 0x10 due to soft errors, memory 0x100 will also contain the corrupted data (0x11). However, corrupted in $r1$ is overwritten by instruction B (MOV $r1$, $r2$), and the corrupted memory 0x100 is also overwritten by instruction C (STORE $r3$, 0x100). Memory 0x100 is updated by instruction A (STORE $r1$, 0x100), but the memory data is overwritten by instruction C (STORE lr, 0x100) before being used or read for program execution. Thus, it does not affect the program if soft error corrupts the data in $r1$ at instruction A (STORE $r1$, 0x100) since the result of instruction A is not used (overwritten).

```
A: STORE r1, 0x100
@  r1: 0x11 (original: 0x10)
@  0x100: 0x11 (original: 0x10)
B: MOV r1, r2
@  Register 1 (r1) is overwritten by r2
C: STORE r3, 0x100
@  Memory data 0x100 is overwritten by r3
```

Figure 3. Exemplary scenario to show software-level masking effects from dynamically dead instructions.

Second, instructions can return correct results even when they read corrupted data. These software-level masking effects can occur in logical instructions (such as logical AND and OR instructions) and compare instructions. If one of the source bits of logical AND instruction is zero, the result is always zero, regardless of the source bit. On the other hand, logical OR instruction always returns one if the source bit of the instruction is one. The result of the comparison instructions can also be correct even if one of the source registers is corrupted. Assume that an instruction compares the data between the two registers, and they have different values. Even though register data is corrupted by soft errors, the values in the two registers can still be different. If a source register of instruction is damaged, we need to trace whether the results of these instructions are different from the original execution.

We have captured and simplified the masking effect at the logical AND instruction from our preliminary experiments. In Figure 4a, a logical AND instruction (AND $r1$, $r2$, $r3$) stores the result of logical AND between data in $r2$ and $r3$ into $r1$ register. In addition, source registers $r2$ and $r3$ store 0x00 and 0x01, respectively. Assume that data contained in $r2$ is corrupted, and it is changed to 0x10 from 0x00. Fault in data contained in $r2$ is not propagated to the destination register ($r1$) since the result of corrupted AND instruction (0x10 AND 0x01) and that of the original one (0x00 AND 0x01) are still identical (0x00). If one of the source bits of logical AND instruction is zero, the result is always zero regardless of another source bit. On the other hand, logical OR instruction always returns one if a source bit of the instruction is one.

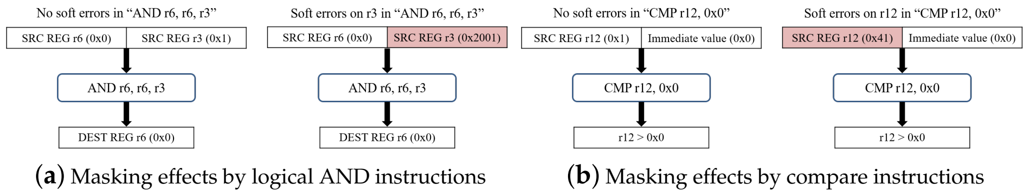


Figure 4. Software-level masking effects analysis from logical and compare instructions.

The following software-level masking effect can occur due to compare instructions, as shown in Figure 4b. Although compare instructions read corrupted data, the result of compare instructions can still be the same. We have found a set of instructions to show the masking effect with compare instructions from our preliminary experiments. In this example, a compare instruction (CMP $r1$, 0x00) compares the data in $r1$ and immediate value zero (0x00). Assume that $r1$ contains 0x01 originally, and it is changed to 0x11 due to soft errors. In case of original execution, data in $r1$ (0x01) is greater than zero (0x00). Even though soft error corrupts data contained in $r1$ to 0x11 from 0x01, data in $r1$ (0x11) is still greater than zero (0x00). Thus, the result of compare instruction (flag bits) can still be the same as compared to that of original execution, and soft errors do not affect the application at all.

Lastly, incorrectly taken branches due to soft errors can still result in the correct output [33]. Wang et al. proposed that approximately 40% of dynamic branches are outcome tolerant, which means that a program can return the correct outputs even though it requires additional instructions due to the incorrect branch. The execution time can be slightly increased or decreased compared to the original execution because control flow violations execute different instructions. Thus, we need to consider two factors from the control flow of a program for software-level masking effects: (i) control flow is correct or not compared to the original execution, and (ii) if an incorrect branch is taken, it affects the final program outputs.

4. Proposed Masking Effects Analysis Framework

4.1. Fault and Failure Model

To characterize system-level masking effects and analyze their distributions, we implemented an automated framework based on fault injection campaigns, as shown in Figure 5. We have performed fault injection campaigns to mimic soft errors in the register file on ARM architecture with gem5 [10] system-call emulation mode. In this work, we have used single-bit flip models instead of multiple-bit soft errors for brevity's sake. The rate of multiple-bit transient faults is increasing with the aggressive technology scaling. However, the rate of multiple-bit error is almost negligible as compared to single-bit errors. For example, the single-bit flip is 100 $\times$ more frequent than multiple-bit soft errors [34].

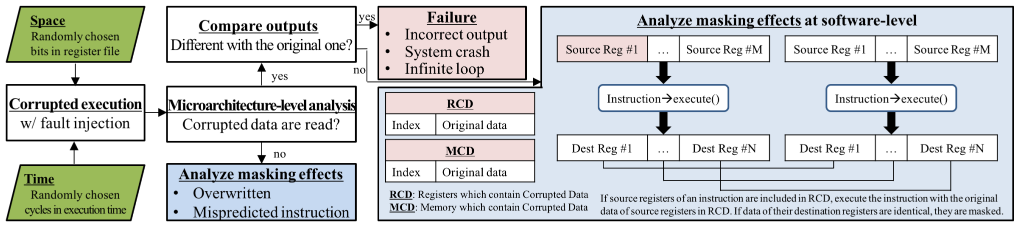

Space
Randomly chosen bits in register file
Corrupted execution
w/ fault injection
Time
Randomly chosen cycles in execution time
Compare outputs
Different with the original one?
yes
no
yes
Microarchitecture-level analysis
Corrupted data are read?
no
Analyze masking effects
• Overwritten
• Mispredicted instruction
Failure
• Incorrect output
• System crash
• Infinite loop
RCD
Index
Original data
MCD
Index
Original data
RCD: Registers which contain Corrupted Data
MCD: Memory which contain Corrupted Data
Analyze masking effects at software-level
Source Reg #1
...
Source Reg #M
Instruction→execute()
Dest Reg #1
...
Dest Reg #N
If source registers of an instruction are included in RCD, execute the instruction with the original data of source registers in RCD. If data of their destination registers are identical, they are masked.

Figure 5. System-level masking effect analysis framework based on exhaustive fault injection campaigns.

Note that we have used system-level gem5 simulator since we can track the detailed behavior of microarchitectural components and software instructions. Therefore, we can track the error propagation of transient faults from hardware and software perspectives. However, it is challenging to capture the detailed behavior of hardware and software by using real hardware. The spatial and temporal domains of the fault injection campaign are randomly chosen from the entire physical integer register and execution time, respectively.

After the randomly selected fault is injected into the register file, our framework traces the microarchitectural behaviors of the microarchitectural components. If committed instructions do not read corrupted data, this fault does not affect the program because they are not used for the program execution, as shown in Figure 2. If committed instructions read corrupted data, we need to trace whether a system crash occurs. If a system crash occurs (e.g., segmentation faults, page table faults, etc.), it is defined as a failure (system crash). It is also defined as a failure if the execution time of the fault-injected execution is more than double the original execution time (i.e., infinite loop). If a program returns the output without a system crash, our framework compares the output with the original one. If their outputs are different, it is also defined as a failure (silent data corruption). If the outputs are still identical to the original one within the deadline, we need to analyze the masking effects at the software level.

4.2. Error Propagation Model

Data structures, RCD (registers that contain corrupted data) and MCD (memory which includes corrupted data), have been exploited to analyze masking effects at the software level, as shown in Figure 5. We traced memory by using MCD and register data by RCD because corrupted data in the register can be propagated to the memory. RCD and MCD store indexes (register number or memory address) contain corrupted data and their original for register and memory, respectively. Note that (index, data) represents the index and original data in RCD and MCD, respectively. The following error propagation model updates the RCD and MCD during corrupted execution.

We modeled error propagation according to various types of instructions. We analyzed the error spread of non-memory instructions, such as data movement between registers (MOV) and ALU instructions. If non-memory instructions read corrupted data, errors can be propagated to the destination register. Assume that there is an addition instruction (ADD *r*1, *r*2, *r*3), and *r*2 and *r*3 contain 0x01 and 0x02, respectively. Without soft errors, data contained in *r*3 should become 0x03 as a result of the addition instruction. If data contained in *r*2 is corrupted and changed to 0x00 from 0x01 because of soft errors, [(*r*2, 0x01)] is stored in RCD. After executing the ADD instruction, data in the destination register (*r*3) are changed to 0x02 from 0x03 due to error propagation. In this example, RCD is updated to [(*r*2, 0x01), (*r*3, 0x03)].

We also modeled the error propagation of memory instructions, such as store and load. For the error propagation of a store instruction (STORE *r*2, 0x100), *r*2 originally contained 0x01 in the previous example. Memory address 0x100 should store 0x01 if there are no soft errors. In this example, RCD contains [(*r*2, 0x01), (*r*3, 0x03)], which means that *r*2 has corrupted data. After executing the store instruction, memory 0x100 also stores corrupted data (0x00) from corrupted data contained in *r*2, and RCD and MCD stores [(*r*2, 0x01), (*r*3, 0x03)] and [(0x100, 0x01)], respectively. Thus, the MCD should be updated by store instructions when they read corrupted data.

If corrupted memory data is read by the load instructions, errors can be propagated to the destination register. A load instruction (LOAD *r*1, 0x100) reads data from memory 0x100, and it makes data contained in *r*1 into 0x01 without soft errors. However, data in memory 0x100 stores corrupted data (0x00) since MCD stores [(0x100, 0x01)]. In this example, data contained in *r*1 is corrupted and changed to 0x00 because of the corrupted data in memory 0x100. After executing the load instructions, RCD and MCD become [(*r*1, 0x01), (*r*2, 0x01), (*r*3, 0x03)] and [(0x100, 0x01)], respectively. After executing these three instructions, errors in *r*2 are propagated to other registers (*r*1 and *r*3) and memory (0x100).

To analyze the masking effects at the software level, we execute an instruction again with the original data and compare their results if the instruction has source registers that are included in the RCD (shaded), as shown in Figure 5. For instance, a logical OR instruction (OR *r*1, *r*2, *r*3) stores the result of the logical OR between data in *r*2 and *r*3 into *r*1. Assume that data contained in *r*2 is corrupted and changed to 0x11 from the original data 0x10, and RCD stores [(*r*2, 0x10)]. Another source register, *r*3, stores 0x01. Our error propagation model considers that errors in data contained in *r*2 can be propagated to the destination register (*r*1) because the logical OR instruction (non-memory instruction) reads corrupted data in *r*2.

Because a source register (*r*2) of the instruction is included in the RCD, we execute the instruction again with the original data in the RCD. In the case of corrupted execution, data contained in *r*1 becomes 0x11 (0x11 OR 0x01) because *r*2 and *r*3 store 0x11 and 0x01, respectively. Our framework changes data contained in *r*2 to 0x10 by loading the original data from RCD, [(*r*2, 0x10)], and data contained in *r*1 becomes 0x11 (0x10 OR 0x01) for the original execution. Our framework compares the results if two versions (corrupted and original) of execution are performed. In this example, the results of corrupted and original executions are exactly the same (0x11). Thus, the destination register of instruction (*r*1) is not updated into the RCD. Our software-level analysis determines that *r*1 is masked by logical OR instruction even though the instruction reads corrupted data from *r*2. Note that the results of the logical instructions may not be masked. Assume that data contained in *r*2 was changed to 0x00 from 0x10. In this case, data in the destination register (*r*1) becomes 0x01 (0x00 OR 0x01) as the result of corrupted execution. Because the result of the corrupted execution (0x01) is different from that of the original execution (0x11), our framework updates RCD to [(*r*2, 0x10), (*r*1, 0x11)].

Further, our data structure can also determine whether soft errors are read by dynamically dead instructions. In the previous example, we hold RCD as [(*r*2, 0x10), (*r*1, 0x11)], which means that *r*2 and *r*1 contain incorrect values as compared to the original execution. Assume that the next instructions to access *r*1 are the data transfer instructions (MOV *r*1, *r*4). In this example, the corrupted value in *r*1 is overwritten because *r*4 does not contain corrupted data. This means that *r*1 is masked by dynamically dead instructions because the result (*r*1) of previous OR instruction (OR *r*1, *r*2, *r*3) is overwritten by another instruction before being used.

5. Experimental Observations

5.1. Experimental Setup

We chose six benchmarks, namely matrix multiplication (matmul), stringsearch, susan, jpeg, bitcount, and sha, mainly from the MiBench benchmark suite [15], for representing various characteristics of applications. First off, we have used matrix multiplication as the reference benchmark since it has no specific characteristics compared to other applications. Secondly, the benchmark stringsearch and sha can represent the sensitivity of control-intensive applications. This is because the benchmark stringsearch is a control-intensive application. On the other hand, the benchmark sha has a small number (just 6%) of the branch instruction. Lastly, the benchmark susan and jpeg are used as multimedia applications, such as encoding/decoding and vision techniques.

For a single set of experiments, we have injected single-bit faults into randomly chosen bits in the register file at a randomly chosen time during the execution time for each benchmark. Furthermore, we have performed this fault injection 1000 times for each benchmark. Therefore, we have performed 6000 fault injection campaigns since we have a total of six benchmarks. Randomly chosen 1000 faults can represent the population with 99% confidence level and 5% error margin based on probability theory [31].

In order to find the appropriate number of faults empirically, we injected faults into a benchmark string search from 1 to 1000 by incrementing one fault, as shown in Figure 6. In Figure 6, X-axis represents the number of injected faults, and Y-axis represents the failure rate, which is the number of failures divided by the number of injected faults. The failure rate varies within just 1% for the benchmark stringsearch after 300 fault injections, which means that 1000 fault injections can represent the entire design space.

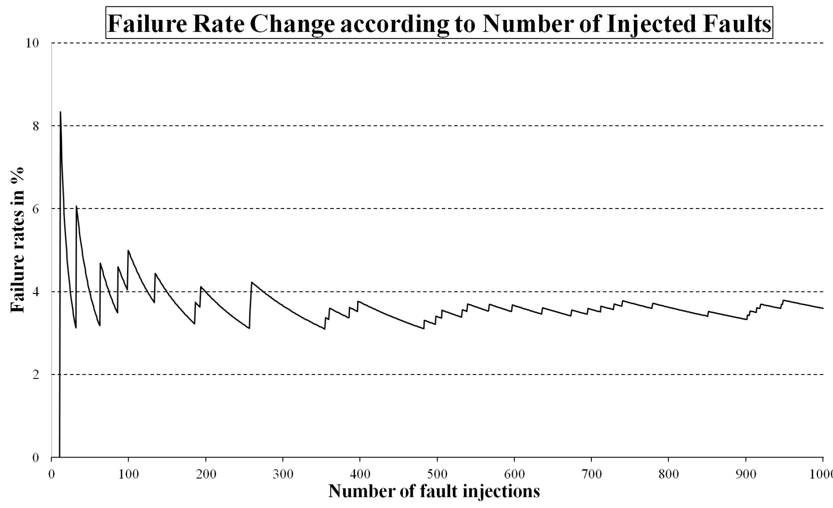
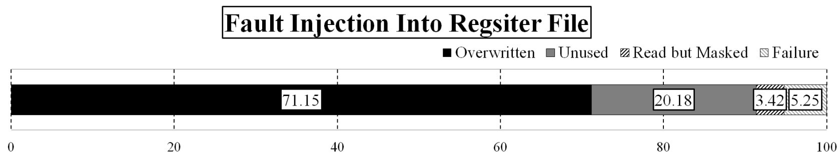


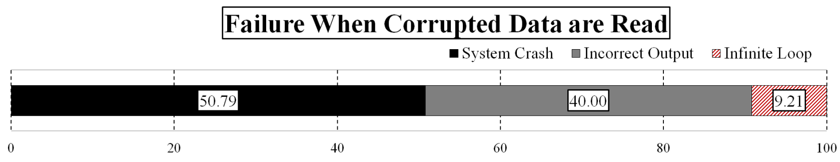
Figure 6. After 300 fault injections, failure rate varies within 1% for a benchmark *stringsearch*.

5.2. Overall Results from Our Masking Analysis Framework

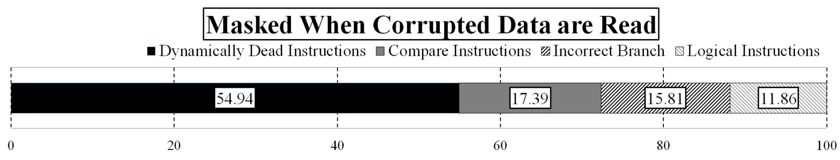
Figure 7 shows the experimental results based on our system-level masking effect analysis framework. Only 5% of soft errors induce system failures, such as infinite loops, system crashes, or incorrect outputs, as shown in Figure 7a. As shown in Figure 7b, approximately 40% of failures are wrong outputs compared to the original ones. For multimedia applications, incorrect outputs can be tolerated if they degrade the quality of the services. For instance, a single-bit flip results in a false output image for a benchmark jpeg, but the difference between the original and faulty images is invisible to the naked eye, as shown in Figure 8. If fault injection campaigns incur system crashes (51%), protection techniques are needed to detect or correct corrupted data to avoid system crashes. In the case of 9% of failures, it loops forever because the loop index values are corrupted due to soft errors.



(**a**) About 95% of injected faults do not affect programs since they are masked at the system-level, such as microarchitecture and software.

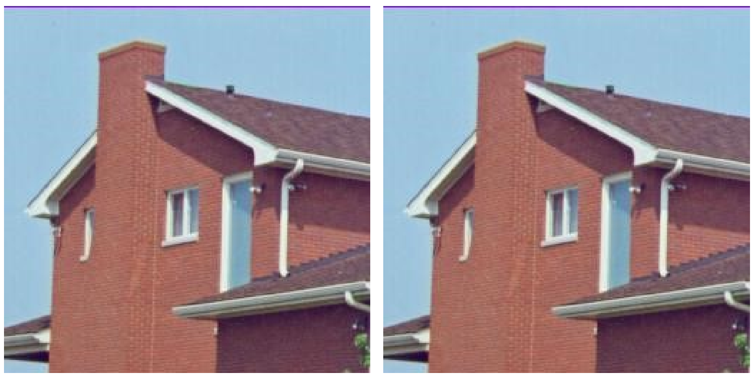


(**b**) Just 5% of injected faults induce system failures, and system crash, incorrect output, and infinite loop take up 51%, 40%, and 9%, respectively.



(**c**) Even though corrupted data is used, they can result in the correct output. About 84% of them do not affect the program execution since they are read by dynamically dead, compare, and logical instructions. In addition, about 16% of them take the incorrect branch, but they return the correct output.

Figure 7. We have injected faults into register file for our set of benchmarks. Interestingly, about 97% of injected faults do not cause failures due to several system-level masking effects.



(**a**) Original output (Size: 9810 bytes) (**b**) Corrupted output (Size: 9819 bytes)

Figure 8. We have injected faults for a benchmark jpeg. Even though faults change the size of the output image, the visual difference between the original and fault image is almost invisible.

As shown in Figure 7a, 91% of the soft errors are masked because committed instructions at the microarchitecture level do not read them. First, 20% of soft errors in register files were not used until the end of the applications. Therefore, it does not change the program execution if there are soft errors in unused registers. Our preliminary experiments showed that approximately 47% of physical registers are not used at all in our benchmark suites. Second, 71% of corrupted data in microarchitectural components can be overwritten by write operations before corrupted data is read or used. For our benchmark suites, 25% of register behaviors are write operations. Third, soft errors in the register file can be masked because of the squashed (i.e., speculative) instructions. Note that not all dynamic instructions are committed in the out-of-order processor due to speculative execution, such as branch prediction.

We analyzed the masking effects at the software level if corrupted data were read by committed instructions, as shown in Figure 7c. Note that a single-bit flip can be masked by several masking effects at the software level. For instance, faults are propagated to another register by data movement instruction, and corrupted data in different registers are masked by compare and logical instructions receptively. More than half of the instances (55%) were masked on average because only dynamically dead instructions read them. Furthermore, 17% and 12% of them are masked by comparison and logical instructions on an average, respectively, because corrupted input data do not change the result of these instructions. Finally, 16% of the instances take the incorrect branch compared to the original fault-free execution, but they can still return the correct outputs. If the wrong branch that does not affect the program output is taken, their execution time varies within 0.18%, on average, compared to the original execution time (at most 2% change for a benchmark stringsearch).

5.3. Detailed Analysis for Software-Level Masking Effects

Figure 9 shows software-level masking effects when committed instructions read corrupted data for our benchmark. As shown in Figure 6, dynamically dead instructions take up most masking effects at the software level for benchmarks, except for susan. Multimedia benchmarks, such as jpeg and susan, have more compare instructions (i.e., conditional statements in C code) than other types of benchmarks due to encoding and edge detection algorithms. Benchmark bitcount, sha, and susan have many logical instructions (especially logical AND statements (&) or logical OR statements (|) in their C codes) in order to count the number of bits of an integer, and they can be masked at the software level. Interestingly, the jpeg and sha benchmarks do not have masking effects from incorrect branch instructions. This is because incorrect branch instructions lead to system failures rather than being masked. For instance, the benchmark sha results in an incorrect output (i.e., silent data corruptions) if an incorrect control flow is taken. The benchmark sha is a secure hash algorithm, and the output can be easily corrupted if the program flow is changed owing to the avalanche effect.

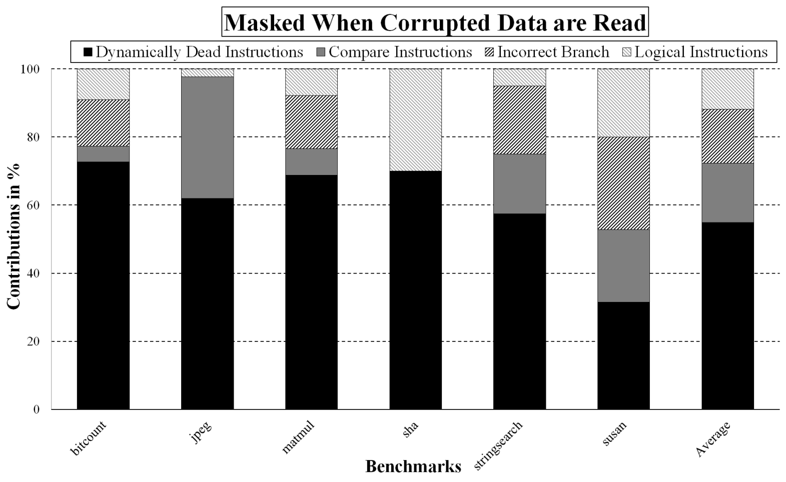


Figure 9. Software-level masking effects for each benchmark when corrupted data is read by committed instructions.

5.4. *Further Analysis for Other Hardware Components*

Our last set of experiments is fault injection into all the microarchitectural components, such as register file, rename map, fetch queue, decode queue, rename queue, I2E (issue to execute) queue, IEW (issue/execute/write-back) queue, instruction queue, reorder buffer, load/store queue for a benchmark, and stringsearch. Figure 10 shows that fault injections into the register file can represent the entire out-of-order processor. In the case of fault injection campaigns to the register file, 93.70% of the injected faults can be masked because the committed instructions do not read them. On the other hand, 91.79% of the injected faults are hidden for the same reason in the case of fault injection into all the microarchitectural components on average. In the case of the register file and all the components, 3.80% and 3.95% of instances did not cause failures, respectively, although committed instructions read corrupted data. Because the difference in distribution between fault injections into the register file and all the components is smaller than 2%, we analyzed system-level masking effects by exhaustive fault injection into the register file.

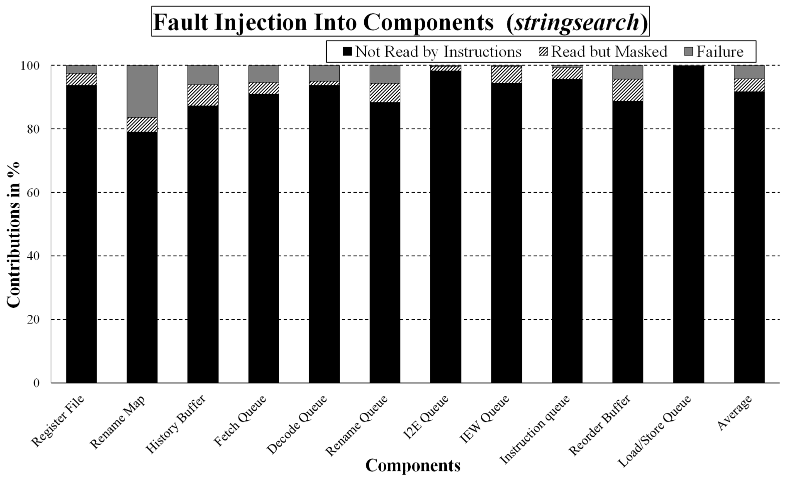


Figure 10. Fault injection campaigns into all the microarchitectural components (Benchmark: stringsearch).

6. Conclusions

Embedded systems suffer from soft errors owing to their tiny feature size and aggressive dynamic voltage and frequency scaling. Several techniques have been proposed to protect embedded processors against soft errors. However, protection techniques against soft errors are not always practical or robust. Thus, we need to quantify the reliability of the system in finding and choosing appropriate protections. Because exhaustive fault injection campaigns and radiation beam testing are too expensive to perform, vulnerability estimation techniques based on cycle-accurate simulators have been proposed. However, vulnerability modeling can be inaccurate owing to several system-level masking effects. Thus, we have characterized the system-level masking effect and have implemented an automated framework to analyze their distribution by exhaustive fault injection campaigns on the register file. In addition, we have examined the microarchitectural behaviors of the register file and found that almost 93% of errors are masked because corrupted data is not read. To analyze software-level masking effects when corrupted data is read by program execution, we also investigated the masking effects at the software level. Based on the software-level analysis, we found that 80% of instances do not change the program at all because faults are injected into dynamically dead, compare, and logical instructions. Approximately 20% of software-level masking effects take the incorrect branch but still produce the correct output by varying the execution time slightly.

We will perform the masking analysis for real hardware devices by exploiting beam testing. Even though the soft error rate is increasing exponentially, a soft error is still a rare event. For example, even though we have tested high-energy neutrons, it still needs more than 500 h to represent 57,000 years in normal execution [35]. Further, even though we have injected radiation-induced faults into hardware devices, it is hard to analyze masking effects since we cannot determine types of soft errors, such as the number of bits and locality of errors. Our future work will also include system-level masking effect analysis for all microarchitectural components in a processor. Thus, our framework will support the comprehensive masking analysis based on various platforms.

Funding: This research received no external funding.

Conflicts of Interest: The author declares no conflict of interest.

References

1. Narayanan, V.; Xie, Y. Reliability Concerns in Embedded System Designs. *Computer* **2006**, *39*, 118–120. [CrossRef]
2. Seifert, N.; Gill, B.; Jahinuzzaman, S.; Basile, J.; Ambrose, V.; Shi, Q.; Allmon, R.; Bramnik, A. Soft Error Susceptibilities of 22 nm Tri-Gate Devices. *IEEE Trans. Nucl. Sci. (TNS)* **2012**, *59*, 2666–2673. [CrossRef]
3. Baumann, R. The impact of technology scaling on soft error rate performance and limits to the efficacy of error correction. In Proceedings of the IEEE International Electron Devices Meeting (IEDM), San Francisco, CA, USA, 8–11 December 2002; pp. 329–332.
4. Ko, Y.; Jeyapaul, R.; Kim, Y.; Lee, K.; Shrivastava, A. Guidelines to Design Parity Protected Write-back L1 Data Cache. In Proceedings of the ACM/EDAC/IEEE Design Automation Conference, San Francisco, CA, USA, 8–12 June 2015; pp. 1–6.
5. Ziegler, J.F.; Muhlfeld, H.P.; Montrose, C.J.; Curtis, H.W.; O'Gorman, T.J.; Ross, J.M. Accelerated Testing for Cosmic Soft-error Rate. *IBM J. Res. Dev.* **1996**, *40*, 51–72. [CrossRef]
6. Hsueh, M.C.; Tsai, T.K.; Iyer, R.K. Fault injection techniques and tools. *Computer* **1997**, *30*, 75–82. [CrossRef]
7. Mukherjee, S.S.; Weaver, C.T.; Emer, J.; Reinhardt, S.K.; Austin, T. Measuring Architectural Vulnerability Factors. *IEEE Micro* **2003**, *23*, 70–75. [CrossRef]
8. Li, X.; Adve, S.V.; Bose, P.; Rivers, J.A. SoftArch: An architecture-level tool for modeling and analyzing soft errors. In Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), Yokohama, Japan, 28 June–1 July 2005; pp. 496–505.
9. Fu, X.; Li, T.; Fortes, J. Sim-SODA: A unified framework for architectural level software reliability analysis. In Proceedings of the International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS), Boston, MA, USA, 18 June 2006.
10. Binkert, N.; Beckmann, B.; Black, G.; Reinhardt, S.K.; Saidi, A.; Basu, A.; Hestness, J.; Hower, D.R.; Krishna, T.; Sardashti, S.; et al. The gem5 Simulator. *ACM SIGARCH Comput. Archit. News* **2011**, *39*, 1–7. [CrossRef]
11. Feng, S.; Gupta, S.; Ansari, A.; Mahlke, S. Shoestring: Probabilistic Soft Error Reliability on the Cheap. In Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), Pittsburgh, PA, USA, 13–17 March 2010.
12. Montesinos, P.; Liu, W.; Torrellas, J. Shield: Cost-effective soft-error protection for register files. In Proceedings of the IBM TJ Watson Conference on Interaction between Architecture, Circuits and Compilers (PAC), Yorktown Heights, NY, USA, October 2006.
13. Kaliorakis, M.; Tselonis, S.; Chatzidimitriou, A.; Gizopoulos, D. Accelerated microarchitectural Fault Injection-based reliability assessment. In Proceedings of the IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFTS), Amherst, MA, USA, 12–14 October 2015; pp. 47–52. [CrossRef]

14. Mirkhani, S.; Cho, H.; Mitra, S.; Abraham, J.A. Rethinking error injection for effective resilience. In Proceedings of the Asia and South Pacific Design Automation Conference (ASP-DAC), Singapore, 20–23 January 2014; pp. 390–393. [CrossRef]
15. Guthaus, M.R.; Ringenberg, J.S.; Ernst, D.; Austin, T.M.; Mudge, T.; Brown, R.B. MiBench: A Free, Commercially Representative Embedded Benchmark Suite. In Proceedings of the IEEE International Symposium on Workload Characterization (IISWC), Austin, TX, USA, 2 December 2001.
16. Lyons, D. SUN screen. *Forbes* **2000**, *166*, 68–70.
17. Junko, Y. Toyota Case: Single Bit Flip That Killed. *EE Times*, 25 October 2013.
18. Baumann, R.C. Radiation-induced soft errors in advanced semiconductor technologies. *IEEE Trans. Device Mater. Reliab.* **2005**, *5*, 305–316. [CrossRef]
19. Baze, M.P.; Buchner, S.P.; McMorrow, D. A digital CMOS design technique for SEU hardening. *IEEE Trans. Nucl. Sci.* **2000**, *47*, 2603–2608. [CrossRef]
20. Krishnaswamy, S.; Plaza, S.M.; Markov, I.L.; Hayes, J.P. Enhancing Design Robustness with Reliability-aware Resynthesis and Logic Simulation. In Proceedings of the IEEE International Conference on Computer-aided Design (ICCAD), San Jose, CA, USA, 4–8 November 2007; pp. 149–154.
21. Lin, S.; Kim, Y.B.; Lombardi, F. Design and Performance Evaluation of Radiation Hardened Latches for Nanoscale CMOS. *IEEE Trans. Very Large Scale Integr. Syst.* **2011**, *19*, 1315–1319. [CrossRef]
22. Li, L.; Degalahal, V.; Vijaykrishnan, N.; Kandemir, M.; Irwin, M.J. Soft Error and Energy Consumption Interactions: A Data Cache Perspective. In Proceedings of the ACM International Symposium on Low Power Electronics and Design (ISLPED), Newport Beach, CA, USA, 9–11 August 2004; pp. 132–137.
23. Manoochehri, M.; Annavaram, M.; Dubois, M. CPPC: Correctable Parity Protected Cache. In Proceedings of the International Symposium on Computer Architecture (ISCA), San Jose, CA, USA, 4–8 June 2011; pp. 223–234.
24. Lee, K.; Shrivastava, A.; Issenin, I.; Dutt, N.; Venkatasubramanian, N. Mitigating Soft Error Failures for Multimedia Applications by Selective Data Protection. In Proceedings of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES), Seoul, Korea, 22–25 October 2006; pp. 411–420.
25. Jeyapaul, R.; Shrivastava, A. Smart cache cleaning: Energy efficient vulnerability reduction in embedded processors. In Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded System, Taipei, Taiwan, 9–14 October 2011; pp. 105–114.
26. Seifert, N.; Tam, N. Timing vulnerability factors of sequentials. *IEEE Trans. Device Mater. Reliab.* **2004**, *4*, 516–522. [CrossRef]
27. Sridharan, V.; Kaeli, D.R. Quantifying Software Vulnerability. In Proceedings of the Workshop on Radiation Effects and Fault Tolerance in Nanometer Technologies (WRET), Ischia, Italy, 5–7 May 2008; pp. 323–328.
28. Borodin, D.; Juurlink, B.H. Protective Redundancy Overhead Reduction Using Instruction Vulnerability Factor. In Proceedings of the ACM International Conference on Computing Frontiers (CF), Bertinoro, Italy, 17–19 May 2010.
29. Goncalves, F.; Santos, M.; Teixeira, I.; Teixeira, J. Self-checking and fault tolerance quality assessment using fault sampling. In Proceedings of the 17th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems, 2002. DFT 2002. Proceedings, Vancouver, BC, Canada, 6–8 November 2002; pp. 216–224. [CrossRef]
30. Sari, A.; Psarakis, M. A fault injection platform for the analysis of soft error effects in FPGA soft processors. In Proceedings of the IEEE 19th International Symposium on Design and Diagnostics of Electronic Circuits Systems (DDECS), Kosice, Slovakia, 20–22 April 2016; pp. 1–6. [CrossRef]
31. Leveugle, R.; Calvez, A.; Maistri, P.; Vanhauwaert, P. Statistical fault injection: Quantified error and confidence. In Proceedings of the Design, Automation and Test in Europe Conference (DATE), Dresden, Germany, 8–12 March 2009.
32. Mukherjee, S.S.; Emer, J.; Reinhardt, S.K. The soft error problem: An architectural perspective. In Proceedings of the IEEE Symposium on High Performance Computer Architecture (HPCA), San Francisco, CA, USA, 12–16 February 2005; pp. 243–247.
33. Wang, N.; Fertig, M.; Patel, S. Y-branches: When you come to a fork in the road, take it. In Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT), New Orleans, LA, USA, 27 September–1 October 2003; pp. 56–66.
34. Lee, K.; Shrivastava, A.; Kim, M.; Dutt, N.; Venkatasubramanian, N. Mitigating the Impact of Hardware Defects on Multimedia Applications: A Cross-Layer Approach. In Proceedings of the 16th ACM International Conference on Multimedia, MM ’08, Vancouver, BC, Canada, 26–31 October 2008; Association for Computing Machinery: New York, NY, USA, 2008; pp. 319–328. [CrossRef]
35. Oliveira, D.; Pilla, L.; DeBardeleben, N.; Blanchard, S.; Quinn, H.; Koren, I.; Navaux, P.; Rech, P. Experimental and Analytical Study of Xeon Phi Reliability. In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC ’17, Denver, CO, USA, 12–17 November 2017; Association for Computing Machinery: New York, NY, USA, 2017. [CrossRef]