electronics

MDPI

Article

Survey of Software-Implemented Soft Error Protection

Yohan Ko

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

Abstract: As soft errors are important design concerns in embedded systems, several schemes have been presented to protect embedded systems against them. Embedded systems can be protected by hardware redundancy; however, hardware-based protections cannot provide flexible protection due to hardware-only protection modifications. Further, they incur significant overheads in terms of area, performance, and power consumption. Therefore, hardware redundancy techniques are not appropriate for resource-constrained embedded systems. On the other hand, software-based protection techniques can be an attractive alternative to protect embedded systems, especially specific-purpose architectures. This manuscript categorizes and compares software-based redundancy techniques for general-purpose and specific-purpose processors, such as VLIW (Very Long Instruction Word) and CGRA (Coarse-Grained Reconfigurable Architectures).

Keywords: soft error; transient fault; fault tolerance; embedded systems; protection technique

Citation: Ko, Y. Survey of Software-Implemented Soft Error Protection. *Electronics* **2022**, *11*, 456. https://doi.org/10.3390/electronics11030456

Academic Editor: Claus Pahl

Received: 31 December 2021
Accepted: 31 January 2022
Published: 3 February 2022

1. Introduction

With technology scaling, the embedded processors' reliability against soft errors is becoming a critical design concern, especially in embedded systems [1]. Soft errors are transient faults in semiconductors caused by external radiations, such as alpha particles, neutrons, muons, and cosmic rays [2]. The soft error rate is constantly increasing [3] and threats to soft errors can no longer be ignored. For instance, a single soft error has stopped a billion-dollar automotive firm every month [4]. Further, the reliability of embedded systems is becoming more critical as embedded systems could be exploited in crucial and safety-critical applications, such as fiscal programs, mobile healthcare devices, and automotive systems, in the near future [5].

As soft errors are hardware-level transient faults in semiconductor devices, diverse hardware-based approaches have been proposed in order to protect systems against soft errors. One of the most straightforward techniques is hardening, that is, making hardware resistant to damage or system malfunctions caused by ionizing radiation. The amount of radiation can be affected by altitude, nuclear energy, and cosmic rays [6]. However, it is impossible to protect systems via hardware hardening perfectly; e.g., neutron-induced soft errors can pass through many meters of concrete [7]. Moreover, hardware hardening techniques also induce severe overheads in terms of area and power consumption.

In order to mitigate overheads, optimized hardware-based protection has been proposed. For memory systems, information redundancy techniques, such as error detection codes (e.g., parity code and Hamming code), have been proposed [8]. Information redundancy schemes detect or correct erroneous data bits by adding check bits to the data based on the coding theory [9]. On the other hand, modular redundancy (e.g., dual or triple modular redundancy) has been presented for non-memory system [10]. Modular redundancy schemes exploit additional modules to detect or correct data mismatch by comparing results between original and replicated modules.

However, it is not surprising that even optimized hardware-based protection techniques still incur significant overheads in terms of hardware costs and power consumption. For instance, the access latency can be larger than the tripled one of unprotected architectures if the level 1 data cache is protected by an error correction code [11]. Even though

hardware-based modular redundancy techniques can execute the same operation in different architectures to minimize the performance overhead, they incur severe overheads in terms of hardware area and power consumption [12].

Hardware-based protection techniques are not appropriate for resource-constrained embedded systems due to overheads in terms of hardware area and power consumption. Further, hardware-based schemes cannot provide knobs to trade reliability for performance as they are applied at the manufacturing stage [13]. In order to mitigate overheads and provide comparable reliability, several software-based techniques have been presented. The origin of software-based protection techniques is primitive. *N*-version programming [14] independently generates *N* functionally equivalent programs and compares their results. Then, it can detect soft errors or software bugs if their results are not identical. However, software engineers always put in considerable effort, as they need to implement the same program in different ways. Thus, there is a necessity to protect embedded systems against soft errors via automated software-based techniques.

In this manuscript, we have categorized software-level protection techniques against soft errors based on the hardware characteristics. Processors can be classified into two major categories such as general-purpose processors and special-purpose processors. As general-purpose processors have no particular target application domain, they can be used for servers, laptops, and tablets. Software-level protection schemes for these processors should provide the comparable fault coverage regardless of ISAs (instruction set architectures). On the other hand, special-purpose processors are used for accelerating kernel parts of programs as co-processors [15]. As these processors can vary depending on their purpose, existing software-based protection schemes do not work in an efficient manner. Thus, various protection schemes have been presented for special-purpose processors considering hardware characteristics.

Oh et al. [16] proposed EDDI (Error Detection by Duplicated Instructions), which duplicates instructions on different registers or memory and inserts comparison and validation at the store or branch operations of the compilation stage. EDDI can detect most soft errors without any hardware modification and additional area overhead. Oh et al. [17] also proposed CFCSS (Control Flow Checking by Software Signature), which monitors the control flow of programs by using assigned signatures in order to protect uncovered parts of EDDI (i.e., control flow violation). Moreover, Reis et al. [18] proposed SWIFT (SoftWare-Implemented Fault Tolerance), adding the optimization techniques to the combination of EDDI and CFCSS. The early stage of software-based duplication and checking techniques, such as EDDI, CFCSS, and SWIFT, can only detect transient faults. These in-thread instruction replication sachems incur massive overheads in terms of performance and power consumption even with several optimization schemes [19,20].

As instruction duplication and control flow checking can only detect soft errors, Reis et al. [21] proposed a software-based error correction technique, SWIFT-R (SWIFT-Recovery). SWIFT-R is a purely software-implemented TMR (triple modular redundancy) with majority voting before critical instructions (e.g., store and branch) and provides near-perfect fault coverage against soft errors. In order to protect only the vital and vulnerable parts of programs instead of all the instructions, Feng et al. [22] proposed a selective instruction duplication scheme by exploiting a vulnerability analysis. Khudia et al. [4] proposed an enhanced approach based on memory profiling in order to apply selective schemes on resource-constrained embedded systems. Profiling-based selective protection schemes provide fault coverage against soft errors comparable to complete instruction duplication, not considering profiling information at the compilation stage.

Software-based redundant multi-threading schemes [23–25] have been presented in order to mitigate the performance overheads from purely software-implemented protections. They execute the same operations on the leading and trailing threads and detect erroneous data by comparing results. However, they also suffer from performance overheads due to frequent synchronization and memory accesses [26,27]. In order to reduce the performance

overheads from inter-thread synchronization, several optimization schemes have been presented for software-level redundant multi-threading schemes [26,27].

However, previous software-based techniques do not consider the characteristics of hardware architectures at all. They can be applied to any kind of processor, such as general-purpose and special-purpose ones, but software-based techniques are not optimized for each hardware. Thus, the optimized protection techniques for special-purpose hardware, such as CGRA (Coarse-Grained Reconfigurable Architecture), VLIW (Very Long Instruction Words), and GPGPU (General-Purpose Computing on Graphics Processing Units), are required for emerging markets. In order to overcome the limitations of software-only approaches, techniques for minimal hardware modification with software protection have been presented. They can also perform complex design space exploration in terms of hardware area, costs, performance, power consumption, and reliability depending on the demand of each application.

The rest of the paper is organized as follows: First, Section 2 presents software-only protection techniques such as instruction duplication, control flow checking and software-implemented error correction schemes. Next, software-based protection techniques that consider hardware characteristics are summarized in Section 3. Finally, Sections 4 and 5 suggest future research directions of software-implemented fault-tolerant techniques and conclude this paper.

2. Purely Software-Implemented Fault-Tolerant Techniques

Oh et al. [16] proposed a simple software-based fault-tolerant technique, EDDI, to detect soft errors without additional hardware modification as described in Table 1. In EDDI, all the instructions are replicated and the comparison instructions are inserted during the compilation stage by using additional resources such as registers and memory. This simple idea can detect most soft errors without any hardware support, but it can incur severe performance overheads due to the expensive duplication and comparison processes. Therefore, in order to minimize performance overhead caused, by comparison, EDDI inserts the comparison instructions right before the store, branch, and jump instructions.

A basic block is defined as the portion of source code with only one entry point and one exit point, i.e., a branch-free sequence of instructions. A storeless basic block never contains the store operation except for the last operation of the basic block. The last operation of a storeless basic block can be a store, branch, or jump operation. In EDDI, the comparison instructions should be inserted just before the last operation of storeless basic blocks. For example, let us assume that the last operation of a storeless basic block is a store operation. Soft errors do not affect the final results unless corrupted data are stored in the memory by the last store operation; thus, the comparison instruction should be placed just before the store operation, rather than before all the instructions, to minimize the performance overhead.

Figure 1b shows the example scenario of EDDI compared with an original code (Figure 1a). In the original code (Figure 1a), the first instruction (1: load r2, (r3)) is a load operation bringing data to r2 from memory address r3. The second instruction (2: add r1, r2, r3) stores the addition between the data stored in r2 and r3 to r1. The last instruction (3: store r1, (r3)) stores r1 data to memory address r3. As shown in Figure 1b, the fourth instruction (4: add r1′, r2′, r3′) is the duplicate of the add instruction. The second instruction (2: load r2′, (r3′+offset)) and the eighth instruction (8: store r1′, (r3′+offset)) are duplicates of the load and store instruction. It is important to add an offset to the memory operation in order to store data in the memory separately. Moreover, the fifth instruction (5: bne r1, r1′, gotoError) and the sixth instruction (6: bne r3, r3′, gotoError) are comparison instructions before the store operation in order to detect soft errors. If the input registers (r1 and r3) of the store instruction are different from their copies (i.e., r1′ r3′), EDDI can detect soft errors.

Table 1. Comparison between software-based protection techniques against soft errors.

Techniques	Key Idea	Pros	Cons
N-Version Programming [14]	Independently generating N > 1 functionally equivalent programs	Reducing the probability of identical software faults	Different programming teams can make similar mistakes
EDDI [16]	Duplicating instructions during compilation and using different registers and variables for the new instructions	Providing high fault coverage Performance overhead can be reduced by using instruction-level parallelism	Huge performance overhead Only detection, not correction No protection mechanism for branch instruction
CFCSS [17]	Monitoring assigned signatures for inter-block control flow checking using instructions	High fault coverage (for control flow checking) Low performance overhead	Only detection, not correction Errors in computation parts cannot be caught by CFCSS
SWIFT [18]	EDDI + ECC (for memory parts) + CFCSS + optimization skills	Performance overhead can be reduced by using instruction-level parallelism	The ratio of correct output is smaller than no protection due to too-conservative error detection Only detection, not correction
SWIFT-R [21]	Intertwining three copies of a program and adding majority voting before critical instructions	Correcting soft errors, not just detecting them	Performance overhead due to instruction triplication
Shoestring [22]	Symptom-based fault detection + software-based instruction duplication	Low performance overhead by selective duplication	Fault coverage of selective protection is not validated
Profile-based solution [4]	Value profiling for generating software symptoms	Low performance overhead by selective duplication Memory profiling can be performed at the compilation stage	Experiments were performed in an in-order processor, not in an out-of-order one Multi-core, multi-thread can change memory profiling methods

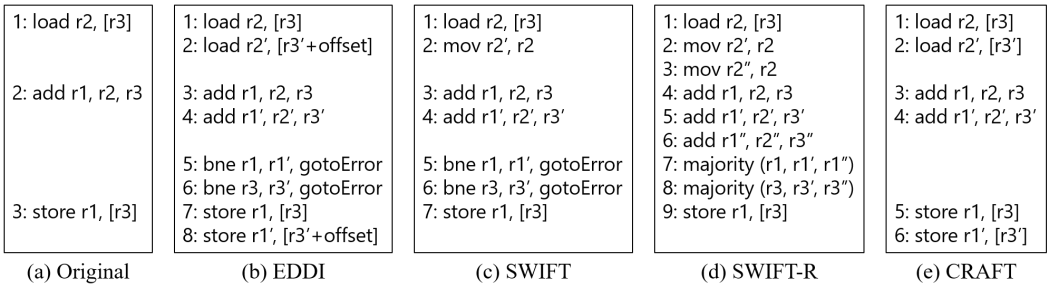

Figure 1. Comparison of instruction stream for several instruction duplication schemes.

Compared to the original instruction stream (three instructions), EDDI (eight instructions) requires five more instructions, including the duplicate and comparison instructions. Thus, it incurs significant overheads in terms of runtime and power. Moreover, EDDI duplicates all the memory instructions, which can incur severe performance overheads. Further, EDDI can only use half of the registers and memory as they need to hold the replicated data for comparison. However, duplicated instructions do not have to be followed by the very master instructions unless instruction scheduling affects the final results. Thus, the performance of instruction duplication can be improved by exploiting instruction-level parallelism. Furthermore, instruction duplication can detect most soft errors as it replicates all the instructions of a program. However, it cannot detect soft errors on the control flow of a program, e.g., EDDI cannot catch a branch incorrectly taken due to soft errors.

Oh et al. [17] proposed a signature monitoring technique, CFCSS, which monitors the control flow of a program using an assigned signature for uncovered parts of EDDI. In CFCSS, the signature is assigned to each basic block to detect soft errors in the control flow. Figure 2 shows an example of how CFCSS works without and with soft errors. V_i, G_i, S_i, and D_i are basic block identifier code, runtime signature, assigned signature for each basic block and pre-calculated difference, respectively. When a program is compiled, the unique signatures S_i are assigned to basic blocks. The pre-calculated differences D_i are also calculated using the XOR operation between connected basic blocks at the compilation stage. Finally, the runtime signature G_i is calculated as the XOR result between the source signature and pre-calculated difference when a program runs.

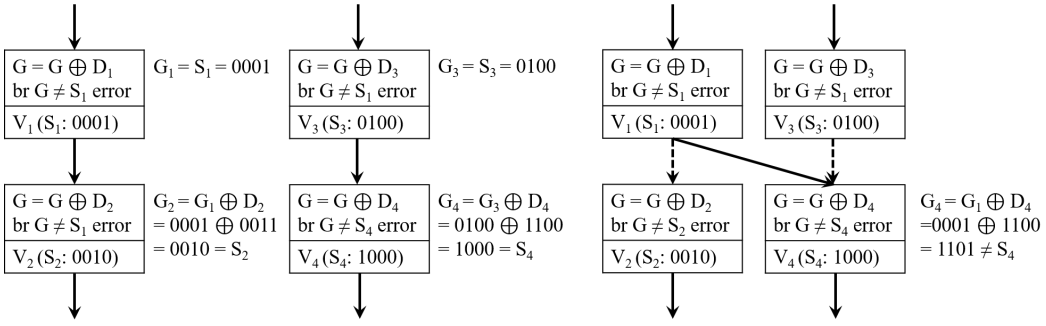

Figure 2. Control flow checking schemes can catch an incorrectly taken branch by using runtime signature and pre-calculated difference.

Let us assume that the original destination basic block from V_1 is V_2 if there is no soft error in the control flow. As shown in Figure 2a, the unique signature of the basic block V_1, S_1, is assigned to 0001, and the unique signature of the correct destination basic block (V_2), S_2, is 0010. Thus, the pre-calculated difference D_2 at the compilation time is 0011, which is the XOR result between S_1 (0001) and S_2 (0010). Moreover, the assigned signatures S_3 and S_4 are 0100 and 1000, respectively. If the basic block V_4 comes from V_3, the pre-calculated difference D_4 is 1100 (0100 XOR 1000). Thus, the runtime signature (G_i) and destination

signature (S_i) should be always exactly the same if there is no soft error in the program flow as shown in Figure 2a.

Let us assume that the destination from V_1 is changed to the basic block V_4 by transient faults. The runtime signature of the basic block V_4, G_4, is calculated as 1101, which is the XOR result between S_1 (0001) and D_4 (1100). As G_4 (1101) is not the same as S_4 (1000), soft errors can be detected by the CFCSS mechanism, as shown in Figure 2b. The CFCSS does not need to modify hardware architectures to detect soft errors in the control flow, as software engineers can allocate unique signatures and runtime signatures to basic blocks by adding codes at the compilation time. However, control flow checking by code addition can incur significant performance overheads. Let us assume that the average number of instructions for each basic block is from 7 to 8 [28]. As we need to add signatures and comparison codes to detect soft errors, we need to add 1 or 2 instructions for every eight instructions at least.

Reis et al. [18] proposed SWIFT, which adds several optimization techniques in combination with EDDI [16] and CFCSS [17]. As the store operation is duplicated using memory offset in EDDI, EDDI can incur overheads in terms of power and performance due to the expensive memory operation. SWIFT eliminates the copy of the store operation shown in Figure Figure 1c, as the memory is protected by error correction codes or other protection techniques in common processors. Instruction duplication and signature-based control flow checking can detect almost 90% of soft errors without any hardware modification.

As SWIFT can only detect soft errors, Reis et al. [21] proposed a software-based recovery technique, SWIFT-R. SWIFT-R triplicates the instructions instead of duplication as in SWIFT. SWIFT only detects soft errors by comparison between the outputs of the original and duplicated instructions. If their results are different, soft errors can be detected. On the other hand, SWIFT-R can correct a soft error by majority voting, as shown in Figure 1d. Even though soft errors corrupt one module, the other two modules still have correct values. Thus, SWIFT-R can be used for safety-critical systems that need near-perfect reliability.

These instruction duplication schemes can protect hardware against soft errors in a simple manner. However, they incur colossal performance overhead as they do not prioritize basic blocks or instructions for duplication. In order to protect only safety-critical or vulnerable parts of applications, Feng et al. [22] proposed Shoestring, a selective instruction duplication scheme, by exploiting a vulnerability analysis. Shoestring has fault tolerance comparable to full instruction duplication, which does not consider profiling information at the compilation stage. Khudia et al. [4] proposed an enhanced Shoestring by adding memory profiling to apply resource-constrained embedded systems.

3. Software-Based Fault-Tolerant Techniques Considering Hardware

Purely software-implemented fault-tolerant techniques can protect hardware without considering hardware characteristics. First, pure software protection can incur severe performance overhead and miss vulnerable cases. In order to improve reliability with comparable area overheads, hybrid fault-tolerant techniques have been presented. Second, previous methods can be applicable regardless of processors, but they can be ineffective for special-purpose processors. In modern embedded systems, many kinds of co-processors, such as VLIW (Very Long Instruction Word) [29], CGRA (Coarse-Grained Reconfigurable Architectures) [30], and GPGPU (General-Purpose computing on Graphics Processing Units) [31], are exploited in order to maximize performance. Thus, optimized software-based approaches have been presented for special-purpose architectures.

Firstly, software-based techniques have been proposed with minimal hardware modification to effectively protect embedded systems against soft errors. Reis et al. [32] proposed CRAFT (CompileR-Assisted Fault Tolerance), which is composed of hybrid hardware/software redundancy techniques as shown in Figure 1d. In this technique, a checking store buffer (CSB) and a load value queue (LVQ) based on SWIFT are added to reduce the number of duplicated store and load instructions and enhance fault coverage; the CSB is a normal store buffer that validates store operations before commit entries. Thus, every

store operation can be validated by comparing original stored data and replicated ones. Moreover, the LVQ performs a comparison between initially loaded data and duplicated loaded ones by using bypasses. Thus, hardware modification has less flexibility than a purely software-based approach, even though its performance is better than software-only fault-tolerant techniques. Therefore, the trade-off among area, reliability, performance, power, and flexibility to implement selective protections is more critical.

Secondly, protection techniques for special-purpose hardware, such as CGRA, VLIW, and GPGPU, are required. Special-purpose architectures accelerate the kernel parts of a program, e.g., simple loop repetition [33]. They can be used as co-processors or accelerators for maximization, and they are drawing significant attention in embedded systems such as signal processing and multimedia computing. However, research on unique architectures has focused on performance improvement, such as parallelization of computation [34] and scheduling algorithms [35], rather than on improving their reliability against soft errors.

As the usage of special-purpose processors is being broadened to critical applications, such as fiscal applications, weather forecasting systems, and ubiquitous medical systems, their reliability is becoming a crucial challenge [36]. Lee et al. [37] and Ko et al. [38] proposed an instruction duplication scheme by exploiting unused slots in VLIW and CGRA architectures with minimal hardware modification. However, they still induced performance overheads even though the purpose of the co-processor and accelerator was performance improvement. Thus, the trade-off relationship between performance and reliability in specific hardware is an essential concern.

4. Discussion

Based on our survey report, there is a necessity for validation of the fault coverage of software-based fault-tolerant techniques against soft errors. As described in Section 2, the duplication or triplication of all instructions without considering the priority of instructions incurs significant overheads in terms of performance and power consumption. Thus, several selective replications with profiling have been proposed in order to reduce performance overheads. Their performance is much better than full duplication as they just duplicate the subset of the full instructions of a program. However, the primary concern of selective protection is to guarantee fault coverage comparable to full protection.

The fault coverage of selective replication has been validated by statistical fault injection [39]. Statistical fault injection injects several faults into microarchitectural components and calculates the fault coverage based on the probabilistic theory. Statistical fault injection needs to inject a lower number of faults to achieve a given confidence and error interval than traditional exhaustive fault injection campaigns. In [22], a single bit-flip fault injection into the register file was implemented to validate their selective protection techniques. As soft errors in the register file can be propagated to other microarchitectural components frequently, fault injection into the register file can effectively show the fault coverage of the entire system.

However, statistical fault injection (SFI) still has three main drawbacks, sampling, availability, and accuracy. For instance, single-bit flips on the register file can provide the fault coverage of the register file against single-bit soft errors. However, it cannot guarantee the fault coverage of other architectures against multi-bit soft errors. This is due to the fact statistical fault injection assumes that the fault model follows the normal distribution, but we cannot ensure whether it really does. Thus, SFI based on the probabilistic theory cannot show the exact fault coverage of selective protections. Moreover, statistical fault injection also has an availability problem as it requires a detailed register transfer level (RTL) model for more accurate modeling. Furthermore, it takes lots of time to inject faults even though RTL models become available. Lastly, SFI cannot provide precise fault coverage against soft errors as it is challenging to mimic realistic soft errors by intentional fault injection campaigns [40].

Therefore, the validation of software-based protection schemes has to choose between accuracy and performance. In order to select accurate fault coverage analysis, fault injection

is not enough, as the soft error is caused by external radiation. Therefore, we have to exploit neutron or electron beam testing to analyze the soft error rate [41]. To the best of our knowledge, there are no research works to compare the fault coverage of various software-level protection schemes. However, beam testing is challenging and time-consuming as it requires expensive equipment and repetitive experiments.

In order to estimate the fault coverage in an efficient manner, alternative fault coverage measurement or estimation will be required to overcome the limitations of conventional fault injection campaigns. Ko et al. [42] estimates the vulnerability of cache memory with and without protection schemes based on the microarchitectural behaviors in order to find better protections in terms of reliability and performance. However, there have been no research works to estimate the fault coverage of software-level schemes based on microarchitectural analysis to the best of our knowledge.

5. Conclusions

In this section, we conclude this paper and present the future research directions of software-based fault-tolerant techniques against soft errors. Soft errors, or transient faults, are becoming a significant design challenge with aggressive technology scaling. In order to eliminate area overheads from hardware-based fault-tolerant techniques, several software-based approaches have been proposed. In this paper, we have demonstrated the development of software-based fault-tolerant techniques against soft errors. In addition, instruction duplication and control flow checking by assigning signatures to basic blocks have been proposed. However, they suffer from performance overheads, as they duplicate all the instructions in a program. Thus, profiling-based selective protection techniques have been proposed to minimize overheads in terms of performance and power consumption.

These pure software schemes do not consider hardware characteristics such as target processors and hardware modification. Firstly, the performance of software-based techniques can be improved by minimal hardware modification, such as using additional memory buffers. Secondly, these techniques can enhance the reliability against soft errors regardless of processors, but they are not optimized for special-purpose processors such as VLIW, CGRA, and GPGPU. Thus, optimized software-based techniques for special-purpose processors have been proposed, e.g., instruction duplication using NOPs on VLIW architectures. Thus, system designers can maximize the effectiveness of software-based fault-tolerant techniques by considering hardware matters.

Funding: This research received no external funding.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Narayanan, V.; Xie, Y. Reliability concerns in embedded system designs. *Computer* **2006**, *39*, 118–120. [CrossRef]
2. Shivakumar, P.; Kistler, M.; Keckler, S.W.; Burger, D.; Alvisi, L. Modeling the effect of technology trends on the soft error rate of combinational logic. In Proceedings of the International Conference on Dependable Systems and Networks, 2002 (DSN 2002), Bethesda, MD, USA, 23–26 June 2002; pp. 389–398. [CrossRef]
3. Fiala, D.; Mueller, F.; Engelmann, C.; Riesen, R.; Ferreira, K.; Brightwell, R. Detection and Correction of Silent Data Corruption for Large-scale High-performance Computing. In Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, Salt Lake City, UT, USA, 10–16 November 2012; IEEE Computer Society Press: Los Alamitos, CA, USA, 2012; SC '12, pp. 78:1–78:12.
4. Khudia, D.S.; Wright, G.; Mahlke, S. Efficient Soft Error Protection for Commodity Embedded Microprocessors Using Profile Information. In Proceedings of the 13th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, Tools and Theory for Embedded Systems, Beijing, China, 12–13 June 2012; ACM: New York, NY, USA, 2012; LCTES '12, pp. 99–108. [CrossRef]
5. Baleani, M.; Ferrari, A.; Mangeruca, L.; Sangiovanni-Vincentelli, A.; Peri, M.; Pezzini, S. Fault-tolerant Platforms for Automotive Safety-critical Applications. In Proceedings of the 2003 International Conference on Compilers, Architecture and Synthesis for Embedded Systems, San Jose, CA, USA, 30 October–1 November 2003; ACM: New York, NY, USA, 2003; CASES '03, pp. 170–177. [CrossRef]

6. Hazucha, P.; Karnik, T.; Walstra, S.; Bloechel, B.A.; Tschanz, J.W.; Maiz, J.; Soumyanath, K.; Dermer, G.E.; Narendra, S.; De, V.; et al. Measurements and analysis of SER-tolerant latch in a 90-nm dual-VT CMOS process. *IEEE J. Solid-State Circuits* **2004**, *39*, 1536–1543. [CrossRef]
7. Mukherjee, S.S.; Emer, J.; Reinhardt, S.K. The soft error problem: An architectural perspective. In Proceedings of the 11th International Symposium on High-Performance Computer Architecture, San Francisco, CA, USA, 12–16 February 2005; pp. 243–247. [CrossRef]
8. Chen, C.L.; Hsiao, M.Y. Error-Correcting Codes for Semiconductor Memory Applications: A State-of-the-Art Review. *IBM J. Res. Dev.* **1984**, *28*, 124–134. [CrossRef]
9. Koren, I.; Krishna, C.M. *Fault-Tolerant Systems*; Morgan Kaufmann: Burlington, MA, USA, 2020.
10. Lyons, R.E.; Vanderkulk, W. The Use of Triple-Modular Redundancy to Improve Computer Reliability. *IBM J. Res. Dev.* **1962**, *6*, 200–209. [CrossRef]
11. Sadler, N.N.; Sorin, D.J. Choosing an Error Protection Scheme for a Microprocessor's L1 Data Cache. In Proceedings of the 2006 International Conference on Computer Design, San Jose, CA, USA, 1–4 October 2006; pp. 499–505. [CrossRef]
12. Shim, B.; Shanbhag, N. Energy-efficient soft error-tolerant digital signal processing. *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.* **2006**, *14*, 336–348. [CrossRef]
13. Martinez-Alvarez, A.; Cuenca-Asensi, S.; Restrepo-Calle, F.; Palomo Pinto, F.R.; Guzman-Miranda, H.; Aguirre, M.A. Compiler-Directed Soft Error Mitigation for Embedded Systems. *IEEE Trans. Dependable Secur. Comput.* **2012**, *9*, 159–172. [CrossRef]
14. Chen, L.; Avizienis, A. N-version programming: A fault-tolerance approach to reliability of software operation. In Proceedings of the Digest of Papers FTCS-8: Eighth Annual International Conference on Fault Tolerant Computing, Toulouse, France, 3–9 June 1978; pp. 3–9.
15. Che, S.; Li, J.; Sheaffer, J.W.; Skadron, K.; Lach, J. Accelerating Compute-Intensive Applications with GPUs and FPGAs. In Proceedings of the 2008 Symposium on Application Specific Processors, Anaheim, CA, USA, 8–9 June 2008; pp. 101–107. [CrossRef]
16. Oh, N.; Shirvani, P.P.; McCluskey, E.J. Error detection by duplicated instructions in super-scalar processors. *IEEE Trans. Reliab.* **2002**, *51*, 63–75. [CrossRef]
17. Oh, N.; Shirvani, P.P.; McCluskey, E.J. Control-flow checking by software signatures. *IEEE Trans. Reliab.* **2002**, *51*, 111–122. [CrossRef]
18. Reis, G.A.; Chang, J.; Vachharajani, N.; Rangan, R.; August, D.I. SWIFT: Software Implemented Fault Tolerance. In Proceedings of the International Symposium on Code Generation and Optimization, San Jose, CA, USA, 20–23 March 2005; IEEE Computer Society: Washington, DC, USA, 2005; CGO '05, pp. 243–254. [CrossRef]
19. Didehban, M.; Shrivastava, A. nZDC: A compiler technique for near Zero Silent Data Corruption. In Proceedings of the 2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC), Austin, TX, USA, 5–9 June 2016; pp. 1–6. [CrossRef]
20. So, H.; Didehban, M.; Jung, J.; Shrivastava, A.; Lee, K. CHITIN: A Comprehensive In-thread Instruction Replication Technique Against Transient Faults. In Proceedings of the 2021 Design, Automation Test in Europe Conference Exhibition (DATE), Grenoble, France, 1–5 February 2021; pp. 1440–1445. [CrossRef]
21. Reis, G.A.; Chang, J.; August, D.I. Automatic Instruction-Level Software-Only Recovery. *IEEE Micro* **2007**, *27*, 36–47. [CrossRef]
22. Feng, S.; Gupta, S.; Ansari, A.; Mahlke, S. Shoestring: Probabilistic Soft Error Reliability on the Cheap. In Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems, Pittsburgh, PA, USA, 13–17 March 2010; ACM: New York, NY, USA, 2010; ASPLOS XV, pp. 385–396. [CrossRef]
23. Wang, C.; Kim, H.S.; Wu, Y.; Ying, V. Compiler-Managed Software-based Redundant Multi-Threading for Transient Fault Detection. In Proceedings of the International Symposium on Code Generation and Optimization (CGO'07), San Jose, CA, USA, 11–14 March 2007; pp. 244–258. [CrossRef]
24. Zhang, Y.; Lee, J.W.; Johnson, N.P.; August, D.I. DAFT: Decoupled acyclic fault tolerance. *Int. J. Parallel Program.* **2012**, *40*, 118–140. [CrossRef]
25. Mitropoulou, K.; Porpodas, V.; Jones, T.M. COMET: Communication-optimised multi-threaded error-detection technique. In Proceedings of the 2016 International Conference on Compliers, Architectures, and Sythesis of Embedded Systems (CASES), Pittsburgh, PA, USA, 2–7 October 2016; pp. 1–10.
26. So, H.; Didehban, M.; Ko, Y.; Shrivastava, A.; Lee, K. EXPERT: Effective and flexible error protection by redundant multithreading. In Proceedings of the 2018 Design, Automation Test in Europe Conference Exhibition (DATE), Dresden, Germany, 19–23 March 2018; pp. 533–538. [CrossRef]
27. So, H.; Didehban, M.; Shrivastava, A.; Lee, K. A software-level Redundant MultiThreading for Soft/Hard Error Detection and Recovery. In Proceedings of the 2019 Design, Automation Test in Europe Conference Exhibition (DATE), Florence, Italy, 25–29 March 2019; pp. 1559–1562. [CrossRef]
28. Hennessy, J.L.; Patterson, D.A. *Computer Architecture: A Quantitative Approach*; Morgan Kaufmann: Burlington, MA, USA, 2011.
29. Faraboschi, P.; Brown, G.; Fisher, J.A.; Desoli, G.; Homewood, F. *Lx: A Technology Platform for Customizable VLIW Embedded Processing*; ACM: New York, NY, USA, 2000; Volume 28.
30. Kim, Y.; Mahapatra, R.N. Hierarchical Reconfigurable Computing Arrays for Efficient CGRA-based Embedded Systems. In Proceedings of the 46th Annual Design Automation Conference, San Francisco, CA, USA, 26–31 July 2009; ACM: New York, NY, USA, 2009; DAC '09, pp. 826–831. [CrossRef]

31. Clemons, J.; Jones, A.; Perricone, R.; Savarese, S.; Austin, T. EFFEX: An embedded processor for computer vision based feature extraction. In Proceedings of the 2011 48th ACM/EDAC/IEEE Design Automation Conference (DAC), San Diego, CA, USA, 5–9 June 2011; pp. 1020–1025.
32. Reis, G.A.; Chang, J.; Vachharajani, N.; Mukherjee, S.S.; Rangan, R.; August, D.I. Design and evaluation of hybrid fault-detection systems. In Proceedings of the 32nd International Symposium on Computer Architecture (ISCA'05), Madison, Wisconsin, 4–8 June 2005; pp. 148–159. [CrossRef]
33. Boyer, M.; Tarjan, D.; Acton, S.T.; Skadron, K. Accelerating leukocyte tracking using CUDA: A case study in leveraging manycore coprocessors. In Proceedings of the IPDPS 2009, IEEE International Symposium on Parallel Distributed Processing, Rome, Italy, 23–29 May 2009; pp. 1–12. [CrossRef]
34. Maitre, O.; Baumes, L.A.; Lachiche, N.; Corma, A.; Collet, P. Coarse Grain Parallelization of Evolutionary Algorithms on GPGPU Cards with EASEA. In Proceedings of the Conference on Genetic and Evolutionary Computation, Montreal, QC, Canada, 8–12 July 2009; pp. 1403–1410.
35. Park, H.; Fan, K.; Mahlke, S.A.; Oh, T.; Kim, H.; Kim, H.S. Edge-centric Modulo Scheduling for Coarse-grained Reconfigurable Architectures. In Proceedings of the International Conference on Parallel Architectures and Compilation Techniques, Toronto, ON, Canada, 25–29 October 2008; pp. 166–176.
36. Tan, J.; Goswami, N.; Li, T.; Fu, X. Analyzing soft-error vulnerability on GPGPU microarchitecture. In Proceedings of the IEEE International Symposium on Workload Characterization, Austin, TX, USA, 6–8 November 2011; pp. 226–235.
37. Lee, J.; Ko, Y.; Lee, K.; Youn, J.M.; Paek, Y. Dynamic Code Duplication with Vulnerability Awareness for Soft Error Detection on VLIW Architectures. *ACM Trans. Archit. Code Optim.* **2013**, *9*, 48. [CrossRef]
38. Ko, Y.; Kang, J.; Lee, J.; Kim, Y.; Kim, J.; So, H.; Lee, K.; Paek, Y. Software-Based Selective Validation Techniques for Robust CGRAs Against Soft Errors. *ACM Trans. Embed. Comput. Syst.* **2016**, *15*, 20. [CrossRef]
39. Leveugle, R.; Calvez, A.; Maistri, P.; Vanhauwaert, P. Statistical fault injection: Quantified error and confidence. In Proceedings of the 2009 Design, Automation Test in Europe Conference Exhibition, Nice, France, 20–24 April 2009; pp. 502–506. [CrossRef]
40. Chatzidimitriou, A.; Bodmann, P.; Papadimitriou, G.; Gizopoulos, D.; Rech, P. Demystifying Soft Error Assessment Strategies on ARM CPUs: Microarchitectural Fault Injection vs. Neutron Beam Experiments. In Proceedings of the 2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), Portland, OR, USA, 24–27 June 2019; pp. 26–38. [CrossRef]
41. Yang, W.; Li, Y.; Zhang, W.; Guo, Y.; Zhao, H.; Wei, J.; Li, Y.; He, C.; Chen, K.; Guo, G.; et al. Electron inducing soft errors in 28 nm system-on-Chip. *Radiat. Eff. Defects Solids* **2020**, *175*, 745–754. [CrossRef]
42. Ko, Y.; Jeyapaul, R.; Kim, Y.; Lee, K.; Shrivastava, A. Protecting Caches from Soft Errors: A Microarchitect's Perspective. *ACM Trans. Embed. Comput. Syst.* **2017**, *16*, 93. [CrossRef]