



## Article

# Revisiting Symptom-Based Fault Tolerant Techniques against Soft Errors

Hwisoo So <sup>1</sup>, Moslem Didehban <sup>2</sup>, Yohan Ko <sup>3,\*</sup>, Reiley Jeyapaul <sup>4</sup>, Jongho Kim <sup>5</sup>, Youngbin Kim <sup>6</sup>,  
Kyoungwoo Lee <sup>1</sup> and Aviral Shrivastava <sup>7</sup>

<sup>1</sup> Department of Computer Science, Yonsei University, 50 Yonsei-ro, Seodaemun-gu, Seoul 03722, Korea; shs7719@yonsei.ac.kr (H.S.); kyoungwoo.lee@yonsei.ac.kr (K.L.)

<sup>2</sup> Mythic, 333 Twin Dolphin DR STE 300, Redwood City, CA 94065, USA; Moslem.Didehban@asu.edu

<sup>3</sup> Division of Software, Yonsei University, Wonju 26493, Korea

<sup>4</sup> ARM, Austin, TX 78735, USA; reiley.jeyapaul@arm.com

<sup>5</sup> TmaxSoft, BS Tower 29, Hwangsaeul-ro 258 beon-gil, Bundang-gu, Seongnam-si 13595, Korea; jongho.kim@yonsei.ac.kr

<sup>6</sup> ETRI, 218 Gajeong-ro, Yuseong-gu, Daejeon 34129, Korea; yb.kim@etri.re.kr

<sup>7</sup> School of Computing, Informatics, and Decision Systems Engineering, Arizona State University, Centerpoint, 660, S Mill Ave., Tempe, AZ 85281, USA; Aviral.Shrivastava@asu.edu

\* Correspondence: yohan.ko@yonsei.ac.kr

**Abstract:** Aggressive technology scaling and near-threshold computing have made soft error reliability one of the leading design considerations in modern embedded microprocessors. Although traditional hardware/software redundancy-based schemes can provide a high level of protection, they incur significant overheads in terms of performance and hardware resources. The considerable overheads from such full redundancy-based techniques has motivated researchers to propose low-cost soft error protection schemes, such as symptom-based error protection schemes. The main idea behind a symptom-based error protection scheme is that soft errors in the system will quickly generate some symptoms, such as exceptions, branch mispredictions, cache or TLB misses, or unpredictable variable values. Therefore, monitoring such infrequent symptoms makes it possible to cover the manifestation of failures caused by soft errors. Symptom-based protection schemes have been suggested as shortcuts to achieve acceptable reliability with comparable overheads. Since the symptom-based protection schemes seem attractive due to their generality and simplicity, even state-of-the-art protection schemes exploit them as the baseline protections. However, our detailed analysis of the fault coverage and performance overheads of such schemes reveals that the user-visible failure coverage, particularly of ReStore, is limited (29% on average). By contrast, the runtime overheads are significant (40% on average) because the majority of the fault injection experiments, which were considered as detected/recovered failures by low-level symptoms, are actually benign faults by program-level masking effects.

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



check for updates

**Citation:** So, H.; Didehban, M.; Ko, Y.; Jeyapaul, R.; Kim, J.; Kim, Y.; Lee, K.; Shrivastava, A. Revisiting Symptom-Based Fault Tolerant Techniques against Soft Errors. *Electronics* **2021**, *10*, 3028. <https://doi.org/10.3390/electronics10233028>

Academic Editors: Zehui Mao, Xinggang Yan and Hamed Badihi

Received: 16 November 2021  
Accepted: 30 November 2021  
Published: 4 December 2021

**Publisher's Note:** MDPI stays neutral with regard to jurisdictional claims in published maps and institutional affiliations.



**Copyright:** © 2021 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<https://creativecommons.org/licenses/by/4.0/>).

## 1. Introduction

Soft errors or transient faults are considered primary sources of unreliability in modern processors. Soft errors caused by external sources, such as high-energy neutrons and alpha particles, or internal events, such as noise in the power voltage, can alter the state of a transistor or change the logical value stored in a memory element of the microprocessor. Despite the existence of several masking effects, ranging from circuit-level [1] to software-level [2,3], some soft errors might not be masked, causing system failures. Traditionally, soft errors have been problematic for high-altitude applications, such as airplanes [4] and space craft [5,6]. However, even ground-level applications have experienced soft error-induced failures owing to sub-nano transistor scaling and near-threshold computing. The International Technology Roadmap for Semiconductors (ITRS) [7] predicted that even

low-energy terrestrial particles, that is, muon particles, will become the main source of soft errors. Nevertheless, because fault-free execution remains common, it is crucial to control the cost (in terms of area, energy consumption, and performance) of soft error protection.

The total cost of a fault-tolerant scheme can be divided into error detection and recovery. The baseline error-tolerant scheme calculates two redundant versions of the computations and checks the results for error detection. If there are any mismatches, a checkpoint and rollback policy can be adopted for recovery purposes. In such schemes, the cost of error detection comes from redundant executions and error checking operations (approximately 100%), and the cost of error recovery comes from preserving the checkpoints for re-execution in the case of an error (we assume  $1 \times$  for frequent checkpointing). Overall, the overheads of the baseline error-tolerant scheme are approximately 200% or more. Whereas software-only schemes impose overheads in terms of the execution time of the programs, hardware redundancy-based schemes incur an area overhead. However, the redundancy-based protection schemes are inappropriate for resource-constrained embedded systems due to severe overheads in terms of performance and hardware area. The considerable overheads of full-redundancy-based schemes have motivated researchers to propose low-cost soft error protection schemes.

Symptom-based fault-tolerant schemes have been proposed to provide a low-cost reliability solution [8–15]. As the main difference between conventional error-tolerant and symptom-based error-tolerant strategies, the former triggers a recovery only in cases in which a manifested error has been detected. Meanwhile, the latter invokes an error recovery routine upon the observation of symptoms, which may even occur in the fault-free run of the program. As the main claim of symptom-based error-tolerant schemes, if there are any soft errors in a microprocessor (i.e., within a window of 100 instructions), a symptom, such as a branch misprediction, ISA-defined exception, or cache/TLB miss, will occur. Therefore, by re-executing a small portion of the instructions upon observing the symptoms, the impact of soft errors can be eliminated from the computations. Several studies [16–20] have considered symptom-based error detection/recovery schemes as effective and low-cost error resilience strategies. For instance, Venkatagiri et al. [16] described symptom-based error detection schemes as “promising owing to their ultra-low cost, which occasionally allows some errors to escape silent data corruptions of SDCs”.

In this study, we conducted a thorough soft error coverage analysis for symptom-based protection schemes [8,11,21–23] from both hardware and software perspectives. Surprisingly, not only is the coverage of such schemes minimal (approximately 29% on average), the associated performance overheads are also considerably high (approximately 40% on average). Further, it is challenging to prevent system failures by tracking variable value changes. It is because system designers need to answer which variables are related to system failures.

Our investigation reveals that the reasons behind such misunderstandings are (1) an immature failure definition and (2) a flawed error coverage metric, which was used in existing symptom-based protection schemes. They consider any corruption in the architectural state of the program (architectural registers and memory), control-flow mispredictions, and latent errors as failures. Moreover, if the architecture detects such an immature failure, the authors consider it an improvement in microprocessor reliability. However, we found that most of these immature failures were benign faults and did not affect the program output because of various program-level masking effects, for example, Y-branches [3] and silent/dead stores [24,25]. For instance, we discovered that more than 55% of dynamic branches in the Mibench [26] test suite programs were Y-branches, and more than 40% of dynamic memory operations were dead. The existence of these program-level masking effects is entirely against the symptom-based solution because there is always a high chance that errors causing branch mispredictions or cache misses will become masked. No reliability is gained in detecting/covering such errors.

The second reason for an overestimation of the ReStore error coverage is the flawed coverage metric used in their approach. In order to quantify the fault coverage of protection

schemes, the accurate way is the beam testing to generate radiation-induced soft errors [27]. However, beam testing takes lots of time, and it is even challenging to set up correctly. Therefore, fault injection, which mimics transient faults, can be an alternative to estimate the fault coverage. As Schirmeier et al. [28] revealed, the error coverage metric usually misrepresents the soft error protection and creates the illusion that the fault-tolerant scheme has improved the system reliability when, in fact, it might have not. To quantify the importance of using the wrong error coverage metric, we calculate the effectiveness of symptom-based protection schemes in the same way as in the original study. In this case, the precise evaluation showed only a 30% failure rate reduction.

## 2. Background and Motivation

Both academia and industry have been dealing with soft error problems for more than half a century. The first widespread indication of soft errors was reported in 1970 when it was demonstrated that high-energy neutrons from cosmic radiation could cause soft errors in memory and combinational circuits [29]. Although advanced FinFET technology has been shown reduced soft error rates against alpha particles [30], the soft error rate trends for FinFET technology still increase because of proton and muon-induced single-event upsets [31].

### 2.1. Protection Schemes against Soft Errors

In order to protect systems against soft errors, diverse hardware-based approaches have been proposed. One of the most straightforward techniques is hardening, making hardware resist the damage or system malfunction caused by ionizing radiation. The amount of radiation can be affected by altitude, nuclear energy, and cosmic ray [32]. However, it is impossible to protect systems by hardware hardening perfectly, e.g., neutron-induced soft errors can pass through many meters of concrete [33]. 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 code (e.g., parity) and error correction code (e.g., Hamming code), have been proposed [34]. On the other hand, modular redundancy (e.g., dual or triple modular redundancy) has been presented for non-memory systems [35].

Many software-based techniques for protecting the computations from soft errors have been proposed. Most of the proposed schemes are redundancy-based solutions, the computations of which are executed redundantly in space or time, and the presence of an error is determined from any discrepancy in the results. It has been shown that the hardware implementation of these simple fault-tolerant strategies is extremely effective, and such techniques have been used in many safety-critical applications [36], such as air traffic control systems [4] and NASA spacecraft [5,6]. However, because the cost of such hardware redundancy-based schemes is considerably high, they cannot be used in many cost-sensitive embedded applications. Software-level redundancy-based schemes shift the cost of using redundant hardware to the software and apply a trade-off between program execution time and extra hardware. Through both radiation-based testing and extensive fault injection campaigns, it has been shown that software-level redundancy-based schemes can achieve a high degree of error coverage [37,38].

However, the main drawback of hardware/software redundancy-based schemes is that they impose a considerably large overhead on the system. For instance, software redundancy-based schemes such as EDDI [39] and nZDC [38] cause a considerable performance overhead (more than double) to the system by applying temporal redundancy to low-level instructions. This considerably high overhead of redundancy-based error-tolerant schemes has motivated researchers to develop low-cost error-tolerant approaches. Existing low-cost error-tolerant schemes can be divided into four main categories: (i) partial-redundancy schemes [40,41], (ii) control-flow checking schemes [42], (iii) vulnerability-reduction schemes [43], and (iv) symptom-based fault-tolerant schemes.

Partial-redundancy schemes protect more important data [40] or duplicate more critical instructions [41] than full-redundancy schemes to mitigate the performance overhead. However, they have limited fault coverage because they only protect a subset of systems. Although control flow checking schemes [42] have been considered reliable protection techniques for detecting control flow violations, a quantitative analysis [44] showed that control flow checking is ineffective and unreliable against soft errors. Vulnerability-reduction schemes [43] use a reliability-driven instruction scheduler at the compiler level to improve the hardware reliability against soft errors. However, their fault coverage is limited because they are not system-level schemes. This study focuses on symptom-based error-tolerant schemes because the fault coverage and effectiveness of symptom-based techniques have yet to be elucidated through a comprehensive analysis.

## 2.2. Symptom-Based Fault Tolerant Schemes

As the key idea of symptom-based fault-tolerant schemes, it is possible to detect the manifestation of errors by monitoring some rare execution phenomena. In general, the existing symptom-based fault-tolerant schemes can be divided into three categories, as described in Table 1: (i) Low-level hardware symptoms for error detection: techniques that monitor low-level hardware symptoms for error detection, (ii) OS-level symptoms for error detection: schemes that utilize OS-level abnormalities for error detection, and (iii) Application-level symptoms for error detection: techniques that explore application-level features for error detection. For error recovery, all such schemes rely on the existence of some type of regular checkpoint support. They can then simply roll back to the last checkpoint and re-execute the instructions.

**Table 1.** Overview of existing symptom-based schemes.

Descriptions	Coverage	Possible Symptoms	Concerns
Low-level hardware symptoms for error detection [8,21]	<ul style="list-style-type: none"> <li>• Soft errors</li> <li>• Low-detection latency</li> </ul>	<ul style="list-style-type: none"> <li>• Branch mispredictions</li> <li>• Exceptions</li> <li>• TLB/cache misses</li> </ul>	<ul style="list-style-type: none"> <li>• Frequent false alarm by natural symptoms</li> <li>• HW support for checkpointing interval of 100 s of instructions</li> </ul>
OS-level symptoms for error detection [22]	<ul style="list-style-type: none"> <li>• Soft errors and hard errors</li> <li>• High-detection latency</li> </ul>	<ul style="list-style-type: none"> <li>• Fatal traps</li> <li>• High OS activity</li> <li>• Hang/segmentation faults</li> </ul>	<ul style="list-style-type: none"> <li>• Offline profiling</li> <li>• Sophisticated HW support for rollback to 10 s of millions</li> </ul>
Application-level symptoms for error detection [11,23]	<ul style="list-style-type: none"> <li>• Soft and hard errors,</li> </ul>	<ul style="list-style-type: none"> <li>• Range-based invariant</li> </ul>	<ul style="list-style-type: none"> <li>• Offline profiling</li> <li>• Application-level modification</li> </ul>

Table 1 summarizes the main features of these techniques. The first symptom-based fault-tolerant scheme is called ReStore [8,21], by which the authors propose that soft error detection is easily achievable by monitoring low-level hardware symptoms. Figure 1 depicts a conceptual diagram for ReStore-protected microprocessor. In such architecture, symptoms such as branch mispredictions, ISA-defined exceptions, and cache misses, can be detected during the execution. If one of these symptoms are detected, it re-executes last  $N$  instructions. As the main argument of research into ReStore, errors leading to failures usually generate symptoms “noisily” and “quickly”. Noisily means that if there is a failure-inducing soft error in a system, it causes abnormal behaviors in the program executions. For instance, a branch misprediction will occur because errors usually alter the control flow of the program. Alternatively, an exception such as an unknown opcode execution divided by a zero exception or illegal memory access may occur. TLB and cache misses can also be considered symptoms because an incorrect data flow can affect the spatial locality. Finally, as a key part of the argument, soft errors usually generate symptoms almost immediately (within 100 instructions) after their occurrence. Therefore, assuming that hardware can provide an effective checkpointing scheme by simply rolling back to the last checkpoint and resuming the execution from there, the effect of soft errors from the system can be eliminated.

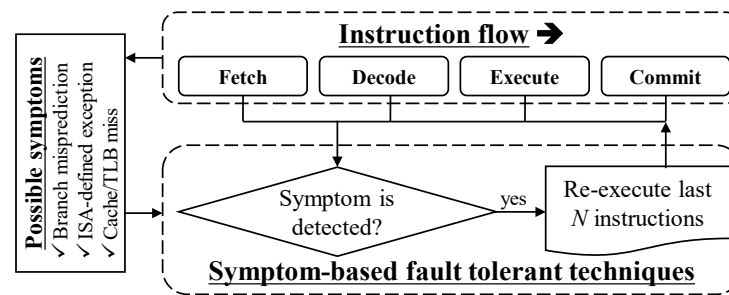
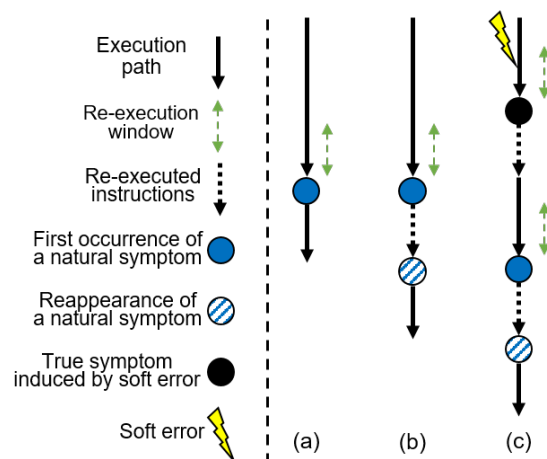


Figure 1. Diagram of symptom-based fault tolerant techniques.

Furthermore, authors studying ReStore have argued that because the hardware structures required for creating low-level checkpointing/rollback mechanisms already exist in the modern speculative processor to handle improper speculations, the ReStore technique can be applied to the existing processors through a minimal hardware modification. As the main drawback of the ReStore scheme, the symptoms are often natural and appear even during the normal execution of the program. For instance, cache misses and branch mispredictions can occur even in a fault-free run of the programs. These natural symptoms can cause a false alarm, leading to an unnecessary rollback and a re-execution of the instructions.

Figure 2 illustrates the execution traces of a program running on a normal and ReStore architecture. Figure 2a shows the program execution trace for a simple system with one natural symptom is shown without any symptom-based protection schemes. Figure 2b illustrates how the execution trace of the program changes on a ReStore-protected machine. As shown, upon observing a symptom, the ReStore architectures re-execute the instructions from the recovery window. If the same symptom reappears, it assumes that the symptom is natural and keeps executing the program. Figure 2c demonstrates the ReStore execution in the presence of a soft error. In this case, the soft error caused a symptom, and the symptom-generation latency was less than that of the re-execution window. Therefore, the ReStore architecture can roll back the execution of the program to a fault-free checkpoint, and it eliminates the effect of the error from the system through a re-execution. Because the symptoms will not be generated again, it is assumed that an error is detected and successfully recovered. However, if the error does not create a symptom or the symptom-generation latency of the error is larger than the re-execution window, the error will remain unrecoverable. Owing to its generality and simplicity, the ReStore strategy has also been used in a hybrid approach (symptom + redundancy), for example, Shoestring [45] and profiling-based soft error protection schemes [46].

OS-level symptom techniques [22] look at the high-level impact of symptoms, such as hardware traps, crashes, and high OS activity, and enhance the fault coverage to hard errors. Rather than detecting low-level symptoms (i.e., exceptions and TLB/cache misses), OS-level symptom techniques postpone error detection until the impact of a fault causes atypical behaviors at the operating system level. Therefore, OS-level symptom techniques trade off the high false alarm rate of low-level symptom-based error detectors with at least a five-orders-of-magnitude longer detection latency. The high error detection latency implies that OS-level symptom techniques require a recovery/rollback strategy to re-execute the last tens of millions of instructions upon observing operating-system-level symptoms. In this case, the overheads of a false alarm are enormous because of the large number of re-executed instructions. Offline program profiling is used to determine the threshold of hang and high OS activity symptoms.



**Figure 2.** (a) Execution traces of a program running unprotected, and a program execution path on a ReStore-protected processor (b) without and (c) with a soft error. Upon the observation of a symptom, the ReStore architecture re-executes the instructions in the re-execution window (last 100 instructions). It ignores the symptom if it happens again; otherwise, it is assumed that a soft error is detected and recovered.

SWAT techniques [23] improve upon OS-level symptom techniques by including likely program-invariant and value-range checking as application-level symptoms. The key idea is to extract the likely values of important program variables by offline profiling and to check whether the dynamically computed value is within the accepted range. The mSWAT framework [11] enhances the SWAT concept for error detection and recovery in multicore systems. The main drawback of OS-level and application-level symptom-detection-based schemes is the lack of generality. Many OS-level symptoms (e.g., hangs, high OS activity, and an acceptable range of variables) are extracted through offline program profiling. They might have difficulty being held under different situations, i.e., changing the input data or using different architectures may show completely different results. Moreover, to replay tens of millions of instructions, sophisticated checkpointing mechanisms that demand major hardware modifications are required.

In order to prove the fault coverage from application-level symptoms by preliminary experiments, we have profiled the value changes of program-defined variables. Then, the authors have picked the specific variables which can affect the program output based on the heuristic way. Lastly, we have added high-level assertions to detect the notable changes of selected variables. For this simple set of experiments, we have chosen a benchmark susan (smoothing). The benchmark handles the multimedia image to smooth the quality, and many variables are limited to the size of the image in ordinary cases. For the specific benchmark susan (smoothing), it can prevent 37% of failures with less than 10% performance overheads. However, the changing value depends on the input. The same benchmark susan (smoothing) cannot detect any failures if we use another set of inputs. Further, it is challenging to select the target variables and their ranges at the profile stage. Thus, we will use the hardware-detectable symptoms in this manuscript.

Even state-of-the-art protection techniques rely on symptom-based techniques due to their simplicity. For example, Minotaur [14] and gem5-Approxilyzer [15] assumed that hardware faults could be detected without severe overheads if they generate the observable symptoms. However, we have to answer the two following questions to exploit symptom-based protections as the baseline guideline. First, how often do failure-inducing soft errors eventually generate symptoms? Secondly, are symptom-based approaches effective in terms of performance? In this paper, we have analyzed the performance effectiveness and reliability improvement of symptom-based protection schemes.

### 2.3. Reconsidering Low-Level Symptoms

OS and application-level symptom-based techniques are limited since they are challenging to apply for the general-purpose processors. On the other hand, low-level symptom-based schemes, mainly ReStore, become an attractive solution for embedded applications due to the high level of generality. However, research into symptom-based protection schemes has not provided any intuition in this area other than fault injection experimental results that explain why a failure-inducing soft error will generate a symptom quickly after its occurrence. We notice the following weaknesses in the experimental results (the sole reason for bounding failures to symptoms), highlighting the need for a precise evaluation of such schemes.

Provably flawed evaluation metric: Traditional symptom-based protection schemes have used the percentage of failures

$$\frac{\text{numbers of faults cause system failures}}{\text{numbers of total fault injection experiments}} \quad (1)$$

as a metric to demonstrate the effectiveness of the scheme. They injected several faults into microarchitectural components, and they counted the number of system failures due to faults. For instance, if 30 faults induce system failures out of totally injected 100 faults, the failure rate will be 30%. However, as a study [28] revealed, such metrics can significantly overestimate the protection capability of schemes such as symptom-based protection schemes, which prolongs the runtime of the program and requires additional hardware.

For example, consider an imaginary fault tolerance scheme that, rather than applying a re-execution of the last 100 instructions on the observations of the symptoms, without reason re-executes the last 100 instructions for some random points of execution. If we randomly inject the same number of faults in the original and imaginary fault-tolerant schemes, the percentage of failures will be reduced. This is because faults inserted close to the randomly selected re-execution point in the imaginary fault-tolerant scheme will become masked, and fewer faults can cause a failure. Therefore, the overall percentage of failures will be improved upon in the baseline architecture, which is an illusion (We strongly encourage the reader to reference [28] for a more detailed explanation of the flaws in the traditional and widely used fault coverage metrics extracted from random fault injection experiments.).

Interestingly, in [18], the authors estimate the coverage of ReStore by both a fault injection campaign and ACE analysis [2]. Because the coverage of the ACE analysis was significantly (around 10×) less than the fault coverage extracted from fault injection results, they improperly conclude that the latter is correct; however, from [28], we know that fault injection results were incorrectly deciphered.

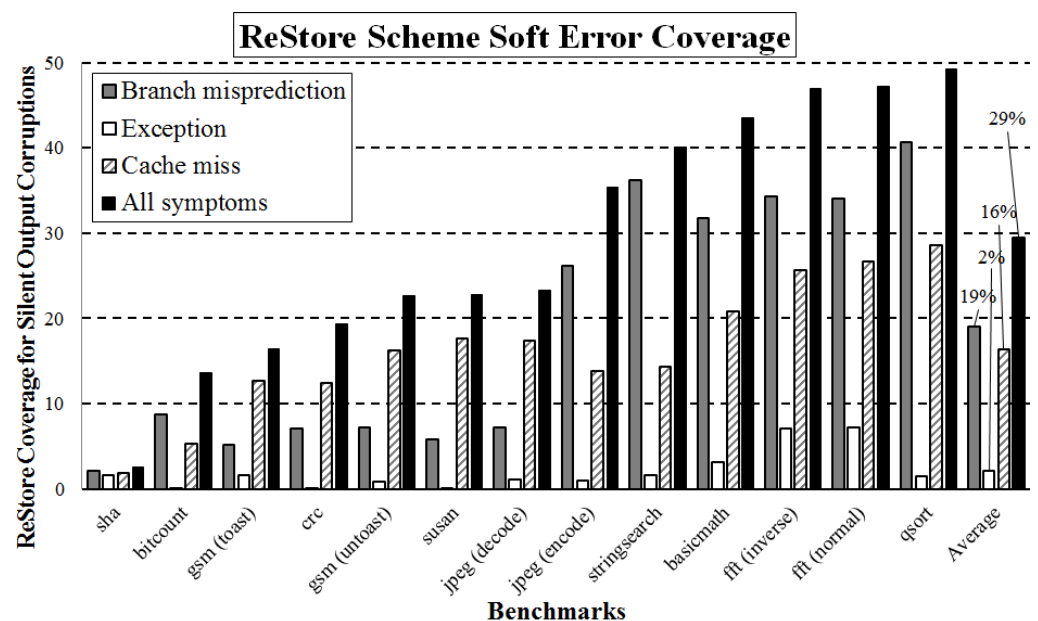
Immature failure definition: Many terms and definitions in studies on ReStore are inconsistent with the widely accepted versions, which have caused frequent misinterpretations regarding the fault coverage capability of the ReStore architecture. For instance, ReStore defines silent data corruptions (SDCs) as cases in which the injected faults corrupt the architectural state of the program (register file and memory state). However, in most studies [12,38,39,45,47], the SDC is considered to be the case in which the error affects the final (user-visible) output of a program [48]. Because several software-level masking effects prevent error propagation from the architectural state to the final output of the program, it is possible that ReStore detects/recovers benign errors (those that eventually do not affect the program outputs). Likewise, ReStore considers all control-flow violations and latent errors as failures. However, as [3] demonstrated, our experimental results also verified that, on average, more than 50% of conditional branches do not affect the correct program behaviors even when forced into incorrect paths. It is also possible that the effects of many latent errors (errors remaining in the system for a long time) will eventually become masked by the program. Therefore, it is crucial to evaluate the protection offered by the ReStore architecture in terms of real user-visible failures. These failures are defined as silent output corruptions (SOCs) [49], a meaningful subset of silent data corruptions.

### 3. Fault Coverage: Overview of Results

To quantify the coverage offered by the architecture covered by symptom-based protection schemes, we supplied a cycle-accurate microarchitecturally simulated microprocessor with perfect branch predictors and caches. There is no branch misprediction, cache miss, or hardware exception in the fault-free runs of programs on the simulated microprocessor. We then injected more than 600,000 single-bit flips on different core components of the simulated processor. We collect information regarding whether the injected fault causes a symptom. If it does, we estimate the distance (in terms of the committed instructions) between the fault injection time and the appearance of the first symptom. Finally, if the symptom-generated fault eventually leads to a user-visible failure, it is defined as a silent output corruption.

Figure 3 demonstrates overall soft error protection coverage of ReStore architecture with the checkpointing interval of 100 instructions. Note that we have exploited the cycle-accurate gem5 simulator [50] to implement the low-level symptom protection scheme, ReStore. We have injected faults that mimic soft errors into the simulator and then estimated the fault coverage of the protection scheme. The Y-axis represents the percentage of silent output corruptions, which can be covered by the ReStore architecture, whereas the X-axis shows programs from the Mibench [26] test suite. As the figure demonstrates, even considering all types of symptoms (the right sidebar for each program), the coverage offered by ReStore is on average approximately 29%. If we consider branch misprediction or cache misses only as symptoms, the average fault coverage is approximately 19% and 16%, respectively. However, because of overlapping symptom phenomena (some faults generate several different symptoms), the overall coverage of ReStore when combining all symptoms is less than the summation of the coverage offered by considering symptoms separately.

As we will discuss later in Section 6.2, the runtime overhead when considering all architecture symptoms is approximately 40%. Overall, we conclude that the assumption that a high level of coverage can be achieved with an extremely low performance overhead by monitoring low-level symptoms is incorrect, as revealed by our comprehensive analysis and exhaustive fault injection campaigns.



**Figure 3.** ReStore architecture can recover just 29% of user-visible failures while considering branch mispredictions, cache misses, and exceptions as symptoms and a recovery window of 100 instructions on average.



#### 4. Fault Coverage Analysis

According to [28], to estimate the protection offered by fault-tolerant schemes, which modify the fault space of a program, the performance and hardware overheads of such a scheme are required. Then, by simply computing the conventional failure rate by conducting random fault injection experiments (percentage of failure-inducing fault injection experiments divided by the total number of fault injection experiments) and multiplying it by the correlation factor  $\gamma$  (essentially the product of the performance and the hardware overheads), the correct failure rate cover can be computed. However, because the failure mentioned in the above rate estimation is heavily implementation dependent, we develop an analytical model for symptom-based protection coverage estimation.

We argue that the protection provided by symptom-based protection schemes is proportional to  $P$ , where  $P$  is the probability that a failure-inducing soft error generates a symptom quickly after its occurrence. In our definition, failure-inducing soft errors are those that modify the user-visible output of the programs. We demonstrate the correctness of our argument for the two cases. Note that we have used three symptoms for detecting soft errors such as cache misses and branch mispredictions as the low-level symptoms [21] and exceptions as the OS-level symptom [22]. For brevity's sake, we call the specific architectures covered by symptom-based protection schemes ReStore. We have assumed ReStore architecture can detect symptoms from various levels, and it can re-execute the latest 100 instructions upon the detection.

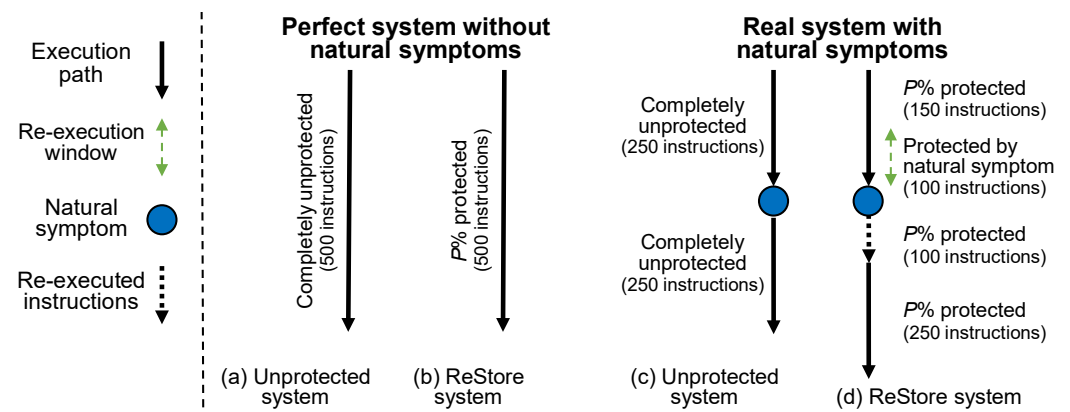
**A perfect system with no natural symptom:** As the ideal case for ReStore coverage, when the branch predictor, as well as caches and TLBs, are perfect, the program execution never exercises any symptoms during its fault-free execution. In this case, there will be no false alarms, and soft errors can be safely considered as the sole reason for any symptoms. By applying ReStore to such an ideal system, however, only soft errors that generate a symptom within the re-execution window size of 100 instructions can be recovered. The ReStore architecture is based on the ability of the system to roll back to a checkpoint and re-execute the last 100 instructions upon the observation of symptoms. If the distance (in terms of instruction) between an error and the generated symptom is larger than the re-execution window size, the error has propagated to the checkpoint, and the re-execution from the checkpoint cannot eliminate the effect of the soft error from the system. In such latent error cases, the symptom will appear again, and ReStore logic considers it a natural symptom. Therefore, even in the case of an ideal branch predictor and cache, the overall protection of ReStore is equal to  $P$ , the probability that a failure-inducing soft error will generate a symptom after its occurrence within a distance less than the checkpointing interval. Note that the results shown in Figure 3 are extracted from fault injection in a symptom-free system based on the same argument. The (a) and (b) parts in Figure 4 demonstrate the execution trace of a program with 500 instructions on an unprotected and ReStore-protected machine.

**A real system with natural symptoms:** As we mentioned before, symptoms do occur even in the fault-free run of the programs. For instance, branch mispredictions and cache misses are part of the executions in real processors. When considering the implementation of the ReStore architecture for a real system, we can assume that the program execution within 100 instructions of natural symptoms is protected (This is an overestimation of fault coverage because (2% in our experimental results) soft errors sometimes occur close to a natural symptom, causing the natural symptom to disappear.). However, we argue that it will not improve the soft error protection in the entire program because the probability of error occurrence before a natural symptom should be considered equal to the probability that an error will occur after the natural symptom (during the re-execution of the last 100 instructions). In other words, as compared to unprotected systems, natural symptoms protect against soft errors that occur within the re-execution window before natural symptoms; however, because they prolong the program execution time, they introduce almost equal soft error vulnerability to the program by a re-execution. Figure 4c,d show traces of a program with 500 instructions and one natural symptom. The entire program execution,

500 instructions, is susceptible to soft errors on an unprotected machine, as shown in part (c). By applying the ReStore scheme to the machine, we can mostly protect the execution time of the 100 instructions before the natural symptom.

Moreover, we also have  $P\%$  protected parts, that is, the re-executed instructions and the rest of the program executions. Note that if we negate the 100 protected instructions from the complete instructions (600 executed instructions on a ReStore-protected machine), there will still be 500 ( $=150 + 100 + 250$ )  $P\%$  protected instructions, which is exactly the same as a system without natural symptoms in part (b) of Figure 4. Natural symptoms impact the coverage of the ReStore scheme both positively and negatively, and overall, it does not affect the total program fault coverage. Therefore, the coverage of the ReStore scheme will not be affected by the natural symptoms, and this case can be simplified as a perfect system, which means that even in cases with natural symptoms, the coverage of the ReStore architecture remains equal to  $P$ .

Overall, by merely estimating the probability of  $P$  for systems with natural symptoms, we can estimate the coverage of the ReStore scheme. We need to estimate  $P$ , which is the probability that a failure-inducing fault will generate a symptom within 100 instructions from its occurrence. If this probability is high, the architecture coverage of ReStore should also be high. Otherwise, the coverage of ReStore is as good as  $P$ .

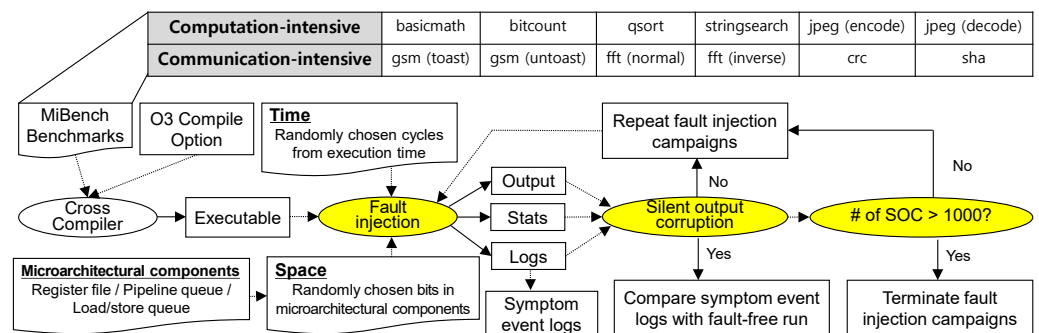


**Figure 4.** Coverage provided by the ReStore architecture on a system without and with natural symptoms. In the first case, the protection of the ReStore architecture is  $P$ , which is the probability that a failure-inducing soft error generates a symptom quickly after its occurrence. Even in the case of systems with natural symptoms, the coverage of ReStore remains the same.

## 5. Experimental Setup

As described in the previous section, we break down the effectiveness of ReStore to the probability  $P$  that a user-visible failure-inducing soft error generates at least one symptom within 100 instructions after its incident. For this purpose, we conducted extensive fault injection experiments on gem5, a cycle-accurate microarchitectural-level simulator [50]. Figure 5 depicts our fault injection and output classification framework. We simulate an ARM Cortex-A53-like processor (The processor is a 32-bit in-order processor with a fixed pipeline, whereas symptom-based fault-tolerant techniques do not depend on the architectures.), which is a modern high-performance and low-power embedded microprocessor. For a ReStore protection estimation, we customized the branch predictor, cache replacement, and prefetching policy for each program so that there are no natural symptoms (branch misprediction and cache miss) in the fault-free run of the programs. We save the fault injection time for each fault injection run, the final output of the program, time, and type of any symptoms if introduced by the injected fault. We use the ARM-GCC 4.6.2 cross compiler with optimization flag O3 to compile benchmarks from the MiBench suite [26]. We categorized MiBench as computation-intensive and communication-intensive applications. The computation-intensive application is composed of numerical calculation (e.g., basicmath, bitcount, qsort, and stringsearch) and image processing (e.g., jpeg encoding

and decoding). Additionally, the communication-intensive application is composed of communication (e.g., gsm and FFT) and security algorithms (sha and crc).



**Figure 5.** Diagram of our fault injection framework. In our campaigns, we gather symptom event logs from more than 1000 silent output corruptions (SOCs) per benchmark through more than 600,000 fault injections.

We have used fault injection to mimic the impact of soft errors since soft errors per bit are very rare in reality [28]. We consider the commonly used single transient bit flip fault as the primary fault model since the majority of system failures is caused by soft errors on the physical register file rather than other microarchitectural components [51]. Further, most soft errors on the microprocessors can eventually modify the state of the physical register file [45]. We randomly selected a fault injection cycle for each fault injection experiment into a randomly chosen bit of the register file. At the end of each fault injection experiment, we classified the outcomes as follows:

**Masked:** The program execution usually terminates, and the final output of the program is correct.

**SOC (Silent Output Corruption):** The program execution usually terminates; however, the final output of the program, compared to that from the fault-free run, is incorrect.

**Others:** The injected fault leads to an early termination of the program execution by causing fatal hardware exceptions, i.e., unknown instruction opcode and illegal access to the memory. It is also defined as “Others” if the processor hangs or the program loops forever.

Because our fault injection experiments aim to quantify the user-level reliability provided by low-level symptom-based error detectors, we focus on the SOC fault injection experiments in this section. In addition, as [28] shows, the number of masked faults is irrelevant to the coverage provided by an error-tolerant scheme. We repeated the fault injection experiments until we collected more than 1000 SOC cases for each benchmark, as shown in Figure 5. Note that because the probability of a fault causing SOC varies with the benchmark applied, the number of fault injection experiments in our framework can differ according to the benchmark. For instance, the SOC rate can be as low as 2% for the suusan benchmark and as high as 50% for the sha benchmark, which is approximately 15% in our experimental results. Consequently, we performed more than 600,000 random fault-injection campaigns for the overall benchmarks.

We explored whether an SOC-inducing fault satisfies the ReStore error coverage condition. If the SOC-inducing fault generates symptoms within 100 instructions, it does; otherwise, it does not. For this, we checked the symptom event log files of the SOC fault injection cases. If there is a record of at least one symptom within 100 instructions after the fault injection time, we consider an SOC-inducing failure as covered by the ReStore architecture. Otherwise, we mark the error as unrecoverable by the ReStore architectures.

## 6. Experimental Results

### 6.1. Ineffective Failure Coverage

Table 2 demonstrates the absolute number of SOC-inducing faults that can be detected/recovered by a different type of low-level symptom in the ReStore architecture with

a checkpoint interval of 100 instructions. Benchmarks are sorted in descending order by the fault coverage of symptom-based protection schemes. As the main point of the results, in most cases (more than 70% of the time on average), SOC-inducing soft errors do not generate a symptom quickly after an incident. This also shows that branch misprediction is the most helpful symptom. On average, approximately 19% of SOC failures can be recovered by simply considering the branch as a hint for error detection. Cache misses are the second most effective symptom. On average, approximately 16% of failures can be avoided if we decipher such events as the presence of errors. However, approximately 6% of the SOC-inducing faults generate both cache misses and branch mispredictions. Note that the hardware exceptions are the most ineffective symptoms, and almost all cases in which soft error induced a hardware exception are also covered with either a branch misprediction or symptoms of a cache miss.

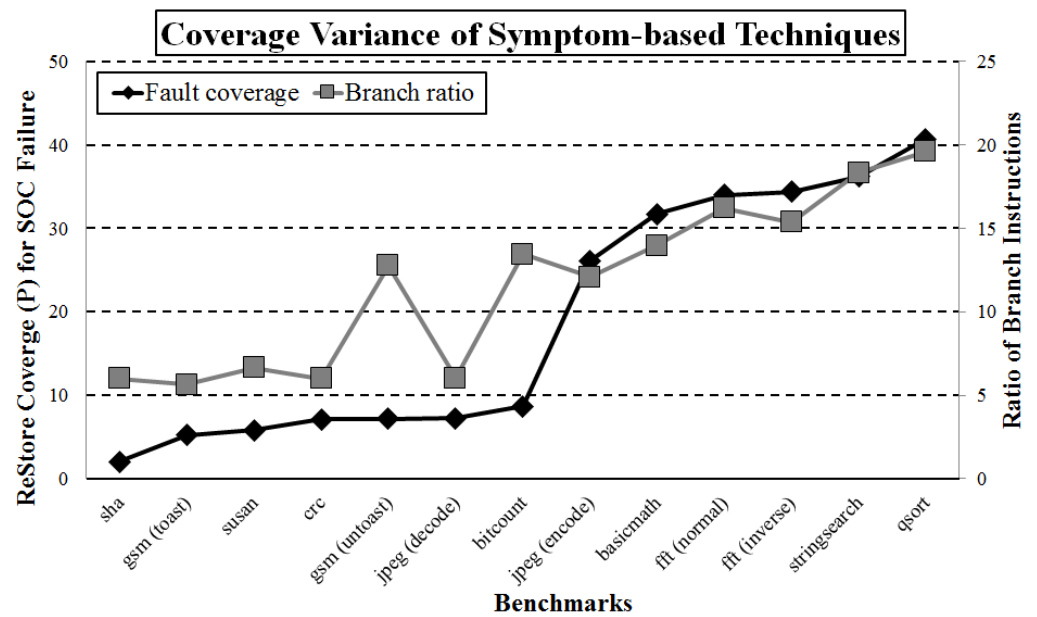
**Table 2.** Number of recoverable SOC-inducing faults (out of 3000) by the ReStore scheme based on the various benchmark.

Benchmarks	BM	EX	CM	All Symptoms
sha	62	49	56	76
bitcount	261	1	158	407
gsm (toast)	157	49	378	493
crc	213	2	371	579
gsm (untoast)	216	27	488	679
susan	175	4	529	683
jpeg (decode)	217	31	523	699
jpeg (encode)	785	27	412	1062
stringsearch	1087	48	431	1203
basicmath	951	93	624	1305
fft (inverse)	1031	212	768	1410
fft (normal)	1021	217	799	1417
qsort	1221	44	858	1478
Average	569	62	492	884

BM: Branch Misprediction, EX: EXception, CM: Cache Miss.

ReStore coverage varies significantly between the different applications. For instance, ReStore performs poorly for the sha benchmark (2%) and performs much better for the qsort benchmark (41%) when we use a branch misprediction as a single symptom. The ratio of branch instructions is one of the main factors that affect the fault coverage of the ReStore scheme, as shown in Figure 6. The ratio of branch instructions was defined as the number of branch instructions over the number of total instructions. Note that the branch instructions include conditional and unconditional control-flow instructions. For a control-intensive benchmark (i.e., many branch instructions in the benchmark), injected faults can cause more branch mispredictions owing to the large number of branch on-demand to instructions are branch instructions, and the fault coverage is only 2% based on branch misprediction symptoms.

By contrast, 20% of the total instructions are branch instructions in the benchmark qsort. Thus, faults are likely to cause branch mispredictions owing to many control-flow instructions, and the fault coverage of the qsort benchmark is larger than those of the other benchmarks. Note that the fault coverage from the cache miss symptoms depends on the ratio of the memory instructions. The ratio of memory instruction for a benchmark qsort is 36%, and the fault coverage achieved from cache miss symptoms is 29%. However, the fault coverage from cache miss symptoms is only 5% for the benchmark bitcount because only 7% of the instructions are memory instructions.



**Figure 6.** ReStore performs well for control-intensive programs, but is ineffective for data-intensive programs.

However, the number of branch instructions in the benchmark determines the coverage. For example, the ReStore fault coverage for benchmarks bitcount and gsm (untoast) is almost similar to that of a benchmark crc, even though the branch instruction ratio of bitcount and gsm (untoast) is much larger than that of crc. Although different applications have a similar ratio of branch instructions, their distributions can be different. To estimate the uniformity of the branch instructions, we estimated the length between consecutive branch instructions. We then calculated the standard deviation of the interval between branch instructions. If the standard deviation is large, the branch instructions are not uniformly distributed. For benchmarks bitcount and gsm (untoast), the standard deviations are 7 and 6, respectively, and their fault coverages achieved from branch mispredictions are close to each other. For a benchmark crc, the standard deviation was only 2. The fault coverage achieved from branch misprediction symptoms is 7%, despite this application having a lower branch instruction ratio than that of bitcount and gsm (untoast).

In general, symptom-based fault-tolerant techniques provide better fault coverage when there are many uniformly distributed symptoms. If there are many symptoms (e.g., a large ratio of branch or memory instructions) and they are uniformly distributed (e.g., with a small standard deviation), the number of instructions between injected faults and symptoms should be small. If the number of symptoms is less than the other benchmarks (e.g., a small ratio of branch or memory instructions), the number of instructions between faults and symptoms can be considerable. If symptoms are not uniformly distributed (e.g., they have a large standard deviation), the instruction length between faults and symptoms can also be greater, despite the large number of symptoms.

### 6.2. Considerable Performance Overhead

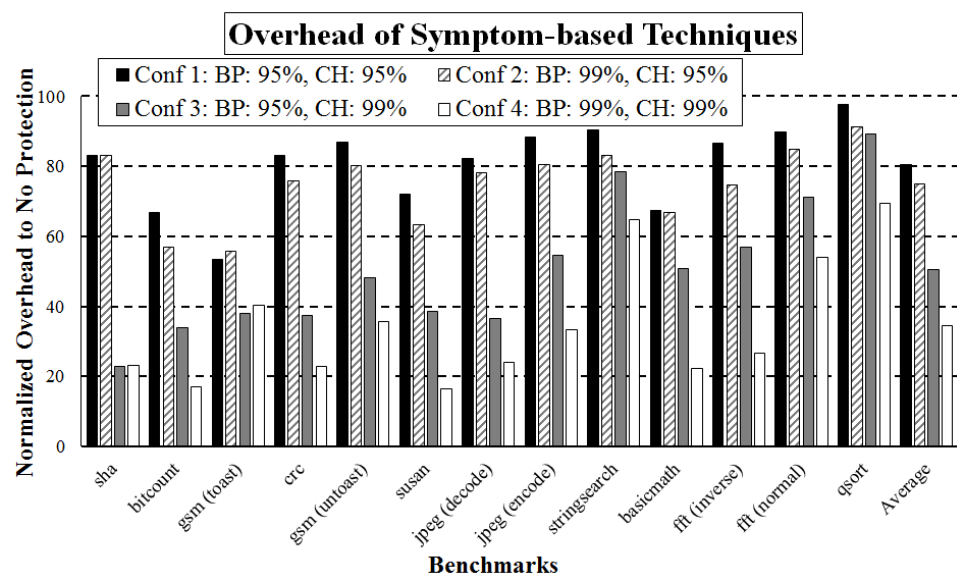
In general, because the symptom-based schemes re-execute some portion of the instructions, e.g., the last  $N$  instructions, under the observation of symptoms, the runtime overhead of such schemes is a function of the frequency of natural symptoms (false alarms) in a system. We use the following equation to estimate the overheads (in terms of extra instructions) in applying the ReStore scheme to a system:

$$\text{Overhead} = (\# \text{ of branch inst} \times \text{branch mispred rate} + \# \text{ of cache accesses} \times \text{cache miss rate}) \times N \quad (2)$$

The above equation simply captures the number of additional instructions that will be re-executed on a symptom-based error coverage scheme where the branch mispredictions

and cache misses are interpreted as signs of soft errors (We do not include exceptions as a symptom here because they are rare, and even re-executing the last  $N$  instructions on the observation of the exceptions does not impose a considerable performance overheads on the system.). However, we should still be careful regarding overlapping symptoms (symptoms with a distance of less than  $N$ ). Basically, if two or more natural symptoms occur in a window size of less than the checkpointing frequency  $N$ , using only Equation (2) may cause a significant overhead overestimation. Therefore, in such cases, for the first symptom, we assume  $N$  instruction overhead, and for the following overlapped symptoms, we only consider their distance (in terms of instructions) from the last symptom as their false alarm overhead. The runtime overheads cannot exceed the runtime of the original programs because we do not re-execute instructions that are already covered by other symptoms.

To collect the information required for the performance overhead estimation of the ReStore architecture, we profiled programs from the MiBench test suite and collected the required statistics, such as the number of branch and memory instructions, branch misprediction, cache miss ratio, and their distribution. Because the performance overheads of the ReStore scheme depend on the branch predictor and cache performance, we compute the overheads of ReStore for four different configurations with different branch predictors and cache efficiencies. We exploit four different hardware architectures by configuring the cache memory and branch predictor to compare the performance overheads due to natural symptoms. We have modified the cache memory and branch predictor to guarantee the certain amount of accuracy. They are categorized as (i) configuration 1, the accuracy of the branch prediction and cache hit ratio of 95% on average; (ii) configuration 2, the accuracy of branch prediction and cache hit ratio of 99% and 95% on average, respectively; (iii) configuration 3, the accuracy of branch prediction and cache hit ratio of 95% and 99% on average, respectively; and (iv) configuration 4, the accuracy of branch prediction and cache hit ratio of 99% on average. Figure 7 demonstrates the performance overhead results for the ReStore scheme with the checkpointing interval of 100 instructions on four different hardware configurations. Interestingly, although we use a 99% accurate branch predictor and cache memory (configuration 4), ReStore increases the runtime by 34% on average. If we use an inaccurate branch predictor and cache memory, which induces many cache misses (configuration 1), the runtime overhead is more than 80%, as compared to without protection.

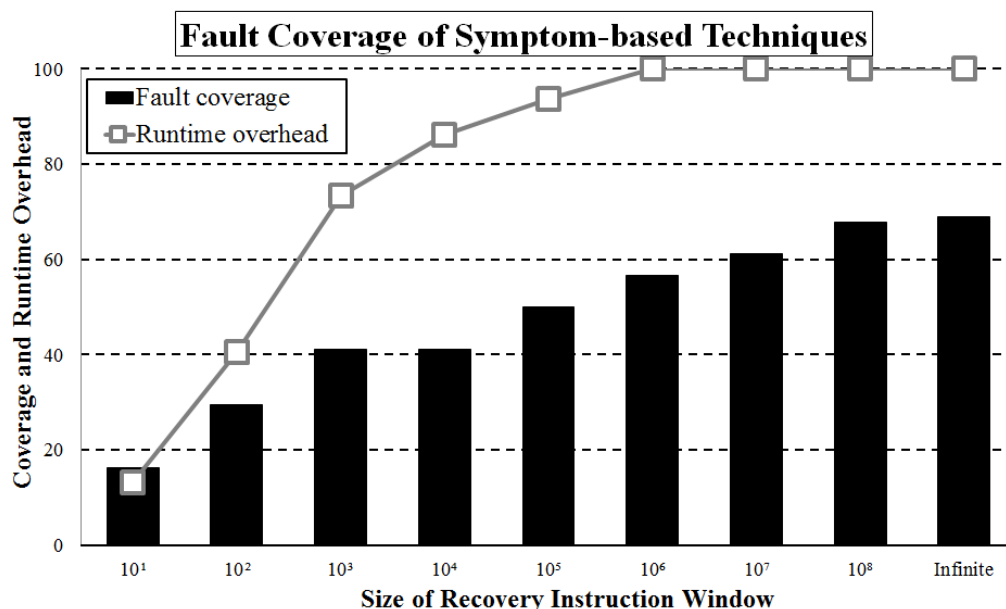


**Figure 7.** Runtime overheads of symptom-based techniques depend on the accuracy of the branch predictor and cache hit ratio. Even though we have exploited 99% accurate branch predictor and cache memory, the performance overhead is more than 40% as compared to unprotected architectures. (Conf, configuration options; BP, accuracy of branch prediction; and CH, cache hit ratio).

To analyze the runtime overheads on the default configuration, we estimated the accuracy of the branch prediction and cache hit ratio. On average, approximately 95% of the predicted branches are correct for our set of benchmarks. Furthermore, the cache hit ratio is more than 98% for all benchmarks, and the average cache hit ratio was almost 100% for our benchmark suite. Symptom-based protection techniques need to re-execute instructions when facing natural symptoms, such as cache misses and branch mispredictions during a fault-free run. Thus, the runtime increases by 40% on average compared to the unprotected architectures. For a benchmark basicmath, the runtime overhead from a branch misprediction as a single symptom is approximately 75%. Branch mispredictions occur for every 70 instructions on average during a fault-free run. The frequency of a branch misprediction is less than 100 instructions, and the runtime overhead is close to the original runtime of the benchmark basicmath. On average, branch mispredictions have the most frequent and heaviest symptoms, followed by cache misses and exceptions.

### 6.3. Recovery Window Size Slightly Improves Coverage and Drastically Hurts Overhead

Figure 8 shows the average fault coverage and runtime overhead of ReStore with different checkpointing intervals or recovery windows. For instance, if the recovery window is less than or equal to 10, 16% of the silent output corruptions can be covered with 13% runtime overhead. In other words, the ReStore architecture can cover or avoid 16% of the silent output corruptions if the hardware provides checkpointing and rollback to the last 10 instructions at any point of the program execution. If system architects want to avoid more than 50% of silent output corruptions, a checkpoint/rollback strategy with the ability to re-execute 100,000 instructions is required; however, the runtime overhead is larger than 94%, as compared to unprotected architectures. Interestingly, approximately 69% of silent output corruptions generate at least one symptom until the end of the application, whereas 31% of silent output corruptions do not generate any symptoms. These 31% of the silent output corruptions cannot be handled or covered by symptom-based fault-tolerant techniques.



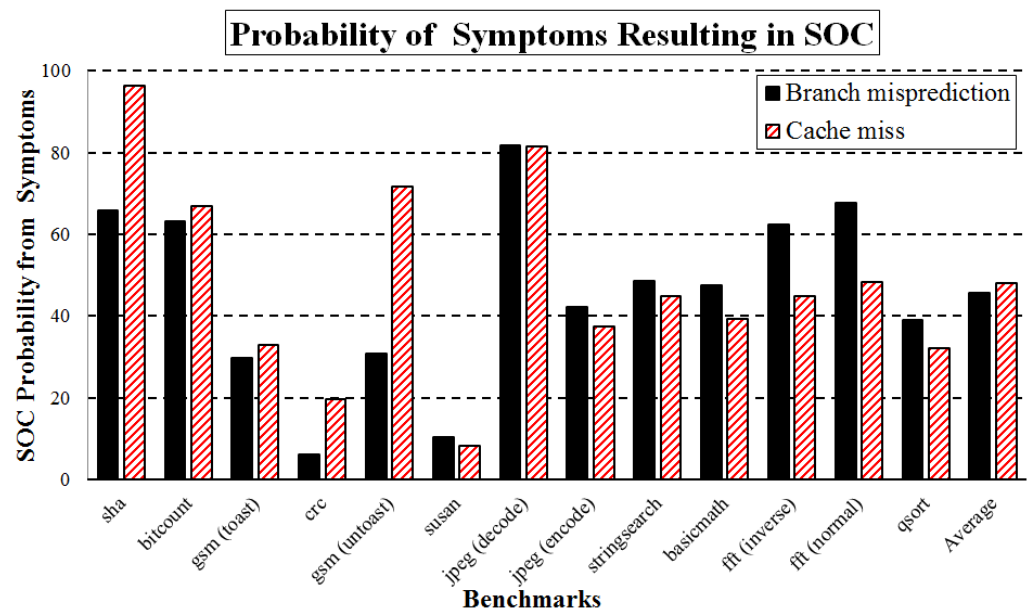
**Figure 8.** We can cover more SOC by re-executing more instructions when symptoms are detected. However, the runtime overhead exceeds the fault coverage if we use a large recovery instruction window.

We also observed that the fault coverage by the recovery instruction window could vary according to the benchmark. We examined the fault coverage of ReStore with different recovery instruction window sizes for two benchmarks, bitcount and gsm (toast). For the bitcount and gsm (toast) benchmarks, only 14% and 16% of the SOC-inducing faults

generate symptoms within 100 instructions, respectively. The benchmark bitcount requires one million instructions of a re-execution for better fault coverage; however, the fault coverage is still 22%. By contrast, the fault coverage of the benchmark gsm (toast) can increase significantly by extending the size of the recovery instruction window. By setting the recovery instruction window size to 1 million, 97% of the silent output corruptions can be covered. Thus, it is difficult to determine the optimal size of the recovery instruction window to satisfy all types of applications.

#### 6.4. Quantifying the Negative Impact of the Program-Level Masking Effects

As we described in Section 2.3, one of the main problems with the ReStore work evaluation is that it did not consider the effect of program-level masking effects on the coverage of ReStore. However, we found that the majority of faults, which generate a symptom soon after their occurrence, will eventually become masked by program-level masking effects. Figure 9 shows this probability separately for a branch misprediction and cache misses. On average, only 46% and 48% of errors detected by a branch misprediction and cache miss symptoms in the ReStore scheme are harmful.



**Figure 9.** Probability of a fault that quickly generates a symptom resulting in SOC is less than half on average. Even symptoms do not strongly imply SOC.

The probability that soft errors will quickly generate symptoms resulting in SOCs depends on the benchmarks used. For instance, Figure 9 shows that only 6% of branch misprediction symptoms cause SOCs for the crc benchmark, whereas more than 82% cause SOCs for the jpeg (decode) benchmark. This occurs because even incorrectly taken branch instructions, called Y-branches, can result in correct outputs, whereas the portion of Y-branches depends on the applications [3]. To analyze the effectiveness of branch misprediction symptoms, we changed the control flow of 300 randomly selected branch instructions over our benchmarks. For the benchmark jpeg (decode), approximately 36% of the control flow violations resulted in correct outputs, whereas the rate was approximately 84% for the crc benchmark. Because the crc benchmark is less sensitive to control flow violations than the jpeg (decode) benchmark, the branch misprediction symptom is not an effective clue for SOCs for crc. Nevertheless, the fact that more than 57% of branches are Y-branches on average demonstrates that a branch misprediction is a poor candidate for representing the existence of soft errors.

The effectiveness of cache miss symptoms is affected by silent/dead memory instructions [24,25], which results in correct outputs even though the memory instructions are



not executed. To analyze the effectiveness of cache miss symptoms, we selected 100 store and load instructions from a benchmark. Moreover, we discarded one of the selected memory instructions for each simulation. For the susan and sha benchmarks, almost 80% and 4% of memory instructions do not affect the program results, respectively, despite not being executed. Approximately 8% of cache miss symptoms induce SOCs for the susan benchmark, as shown in Figure 9, because many memory instructions are not critical in this benchmark. By contrast, 96% of the symptoms cause SOCs for the benchmark sha because most memory instructions are not silent/dead. On average, 48% of memory instructions do not cause failures at all, despite not being executed over our benchmarks, which induces an overprotection of the cache miss symptoms.

## 7. Conclusions

With aggressive technology scaling, the soft error rate is increasing, particularly in modern embedded systems. In order to protect embedded systems against soft errors, several hardware and software redundancy schemes have been proposed. However, they can be expensive in terms of performance and hardware, and they are not suitable for resource-constrained embedded systems. Symptom-based techniques have been suggested as an alternative to protect embedded processors effectively. As the main claim behind symptom-based techniques, soft errors generate symptoms quickly when soft errors cause a failure. Failures can then be avoided by re-executing the last 100 instructions when symptoms are detected. Symptom-based fault-tolerant techniques seem compelling because they do not have to duplicate all instructions or require expensive hardware modifications. Since the symptom-based protection schemes seem attractive due to their generality and simplicity, even state-of-the-art protection schemes exploit them as the baseline protections. In this work, we have implemented the reliability analysis module to reconsider the existing protection schemes. Then, we have found that there are no royal roads to achieve the high reliability. Our experimental results show that symptom-based techniques can cover only 29% of silent output corruptions. Furthermore, their runtime overhead is almost 40% compared to unprotected architectures owing to frequent false alarms caused by natural symptoms. Finally, symptom-based fault-tolerant techniques are ineffective because more than half of the quickly generated symptoms do not cause failures owing to several masking effects.

Our future work will include implementing a more general framework to analyze the efficacy of existing protection schemes since this work focuses on low-level symptom-based protection schemes. Our detailed experimental results found that selective protection schemes need to be validated thoroughly and comprehensively. Then, the analysis will be the first step to reach the reliability goal.

**Author Contributions:** Conceptualization, M.D., Y.K. (Yohan Ko) and R.J.; Data curation, J.K. and H.S.; Writing—original draft, H.S. and Y.K. (Younbin Kim); Writing—review and editing, Y.K. (Yohan Ko); supervision, K.L.; project administration, A.S. All authors have read and agreed to the published version of the manuscript.

**Funding:** This research received no external funding.

**Conflicts of Interest:** The authors declare no conflicts of interest.

## References

1. 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 IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), Washington, DC, USA, 23–26 June 2002; pp. 389–398.
2. Mukherjee, S.S.; Weaver, C.; Emer, J.; Reinhardt, S.K.; Austin, T. A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor. In Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture, San Diego, CA, USA, 5 December 2003; pp. 29–40.
3. Wang, N.; Fertig, M.; Patel, S. Y-branches: When you come to a fork in the road, take it. In Proceedings of the IEEE International Conference on Parallel Architecture and Compilation Techniques (PACT), New Orleans, LA, USA, 27 September–1 October 2003; pp. 56–66.

4. Cristian, F.; Dancey, B.; Dehn, J. Fault-tolerance in the advanced automation system. In Proceedings of the ACM SIGOPS European Workshop, Bologna, Italy, 3–5 September 1990; pp. 6–17.
5. LaBel, K.A.; Barnes, C.E.; Marshall, P.W.; Marshall, C.J.; Johnston, A.H.; Reed, R.A.; Barth, J.L.; Seidleck, C.M.; Kayali, S.A.; O'Bryan, M.V. A roadmap for NASA's radiation effects research in emerging microelectronics and photonics. In Proceedings of the 2000 IEEE Aerospace Conference, Big Sky, MT, USA, 25 March 2000; Volume 5.
6. Katz, D.S.; Some, R.R. NASA advances robotic space exploration. *Computer* **2003**, *36*, 52–61. [[CrossRef](#)]
7. IRC. International Technology Roadmap for Semiconductors 2.0-Executive Summary. 2015. Available online: <http://www.itrs2.net/itrs-reports.html> (accessed on 2 December 2021).
8. Wang, N.J.; Patel, S.J. ReStore: Symptom-based soft error detection in microprocessors. *IEEE Trans. Dependable Secur. Comput. (TDSC)* **2006**, *3*, 188–201. [[CrossRef](#)]
9. Sahoo, S.K.; Li, M.L.; Ramachandran, P.; Adve, S.V.; Adve, V.S.; Zhou, Y. Using likely program invariants to detect hardware errors. In Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), Anchorage, AK, USA, 24–27 June 2008; pp. 70–79.
10. Yalcin, G.; Unsal, O.S.; Cristal, A.; Hur, I.; Valero, M. SymptomTM: Symptom-based error detection and recovery using hardware transactional memory. In Proceedings of the IEEE International Conference on Parallel Architecture and Compilation Techniques (PACT), Galveston, TX, USA, 10–14 October 2011; pp. 199–200.
11. Hari, S.K.S.; Li, M.L.; Ramachandran, P.; Choi, B.; Adve, S.V. mSWAT: Low-cost hardware fault detection and diagnosis for multicore systems. In Proceedings of the 2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), New York, NY, USA, 12–16 December 2009; pp. 122–132.
12. Li, M. Designing Resilient Hardware by Treating Software Anomalies. Ph.D. Thesis, University of Illinois Urbana-Champaign, Champaign, IL, USA, 2009.
13. Ramachandran, P. Detecting and Recovering from In-Core Hardware Faults through Software Anomaly Treatment. Ph.D. Thesis, University of Illinois Urbana-Champaign, Champaign, IL, USA, 2011.
14. Mahmoud, A.; Venkatagiri, R.; Ahmed, K.; Misailovic, S.; Marinov, D.; Fletcher, C.W.; Adve, S.V. Minotaur: Adapting software testing techniques for hardware errors. In Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, Providence, RI, USA, 13–17 April 2019; pp. 1087–1103.
15. Venkatagiri, R.; Ahmed, K.; Mahmoud, A.; Misailovic, S.; Marinov, D.; Fletcher, C.W.; Adve, S.V. gem5-Approxilyzer: An open-source tool for application-level soft error analysis. In Proceedings of the 2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), New York, NY, USA, 12–16 December 2019; pp. 214–221.
16. Venkatagiri, R.; Mahmoud, A.; Hari, S.K.S.; Adve, S.V. Approxilyzer: Towards a systematic framework for instruction-level approximate computing and its application to hardware resiliency. In Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO), Taipei, Taiwan, 15–19 October 2016; pp. 1–14.
17. Feng, S.; Gupta, S.; Ansari, A.; Mahlke, S.A.; August, D.I. Encore: Low-cost, fine-grained transient fault recovery. In Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO), Porto Alegre, Brazil, 3–7 December 2011; pp. 398–409.
18. Wang, N.J.; Mahesri, A.; Patel, S.J. Examining ACE analysis reliability estimates using fault-injection. In Proceedings of the 34th Annual International Symposium on Computer Architecture, San Diego, CA, USA, 9–13 June 2007; Volume 35, pp. 460–469.
19. Snir, M.; Wisniewski, R.W.; Abraham, J.A.; Adve, S.V.; Bagchi, S.; Balaji, P.; Belak, J.; Bose, P.; Cappello, F.; Carlson, B.; et al. Addressing failures in exascale computing. *Sage Int. J. High Perform. Comput. Appl. (IJHPCA)* **2014**, *28*, 129–173. [[CrossRef](#)]
20. Sastry Hari, S.K.; Venkatagiri, R.; Adve, S.V.; Naeimi, H. GangES: Gang error simulation for hardware resiliency evaluation. *ACM Sigarch Comput. Archit. News* **2014**, *42*, 61–72. [[CrossRef](#)]
21. Wang, N.J.; Patel, S.J. ReStore: Symptom based soft error detection in microprocessors. In Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), Yokohama, Japan, 28 June–1 July 2005; pp. 30–39.
22. Li, M.L.; Ramachandran, P.; Sahoo, S.K.; Adve, S.V.; Adve, V.S.; Zhou, Y. Understanding the propagation of hard errors to software and implications for resilient system design. *ACM Sigarch Comput. Archit. News* **2008**, *36*, 265–276. [[CrossRef](#)]
23. Li, M.L.; Ramachandran, P.; Sahoo, S.K.; Adve, S.V.; Adve, V.S.; Zhou, Y. SWAT: An error resilient system. In Proceedings of the Workshop on Silicon Errors in Logic (SELSE), Austin, TX, USA, 26–27 March 2008.
24. Lepak, K.M.; Lipasti, M.H. Silent stores for free. In Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO), Monterey, CA, USA, 10–13 December 2000; pp. 22–31.
25. Bell, G.B.; Lepak, K.M.; Lipasti, M.H. Characterization of silent stores. In Proceedings of the IEEE International Conference on Parallel Architecture and Compilation Techniques (PACT), Philadelphia, PA, USA, 15–19 October 2000; pp. 133–144.
26. 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 Workshop on Workload Characterization (WWC), Austin, TX, USA, 2 December 2001; pp. 3–14.
27. Dixit, A.; Wood, A. The impact of new technology on soft error rates. In Proceedings of the 2011 International Reliability Physics Symposium, Monterey, CA, USA, 10–14 April 2011.
28. Schirmeier, H.; Borchert, C.; Spinczyk, O. Avoiding pitfalls in fault-injection based comparison of program susceptibility to soft errors. In Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), Rio de Janeiro, Brazil, 22–25 June 2015; pp. 319–330.

29. May, T.C.; Woods, M.H. Alpha-particle-induced soft errors in dynamic memories. *IEEE Trans. Electron Devices (T-ED)* **1979**, *26*, 2–9. [[CrossRef](#)]
30. Lee, S.; Kim, I.; Ha, S.; Yu, C.S.; Noh, J.; Pae, S.; Park, J. Radiation-induced soft error rate analyses for 14 nm FinFET SRAM devices. In Proceedings of the IEEE International Reliability Physics Symposium (IRPS), Monterey, CA, USA, 19–23 April 2015.
31. Hubert, G.; Artola, L.; Regis, D. Impact of scaling on the soft error sensitivity of bulk, FDSOI and FinFET technologies due to atmospheric radiation. *Integr. VLSI J.* **2015**, *50*, 39–47. [[CrossRef](#)]
32. Hazucha, P.; Karnik, T.; Walstra, S.; Bloechel, B.A.; Tschanz, J.W.; Maiz, J.; Soumyanath, K.; Dermer, G.E.; Narendra, S.; De, V.; Borkar, S. 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](#)]
33. 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](#)]
34. 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](#)]
35. Lyons, R.E.; Vanderkulk, W. The Use of Triple-Modular Redundancy to Improve Computer Reliability. *IBM J. Res. Dev.* **1962**, *6*, 200–209. [[CrossRef](#)]
36. Ibrahim, Y.; Wang, H.; Bai, M.; Liu, Z.; Wang, J.; Yang, Z.; Chen, Z. Soft error resilience of deep residual networks for object recognition. *IEEE Access* **2020**, *8*, 19490–19503. [[CrossRef](#)]
37. Quinn, H.; Baker, Z.; Fairbanks, T.; Tripp, J.L.; Duran, G. Software resilience and the effectiveness of software mitigation in microcontrollers. *IEEE Trans. Nucl. Sci. (TNS)* **2015**, *62*, 2532–2538. [[CrossRef](#)]
38. Didehban, M.; Shrivastava, A. nZDC: A Compiler technique for near zero silent data corruption. In Proceedings of the ACM/EDAC/IEEE Design Automation Conference (DAC), Austin, TX, USA, 5–9 June 2016; p. 48.
39. Oh, N.; Shirvani, P.P.; McCluskey, E.J. Error detection by duplicated instructions in super-scalar processors. *IEEE Trans. Reliab. (TR)* **2002**, *51*, 63–75. [[CrossRef](#)]
40. Lee, K.; Shrivastava, A.; Issenin, I.; Dutt, N.; Venkatasubramanian, N. Partially protected caches to reduce failures due to soft errors in multimedia applications. *IEEE Trans. Very Large Scale Integr. Syst. (TVLSI)* **2009**, *17*, 1343–1347.
41. 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. (TACO)* **2013**, *9*, 1–24. [[CrossRef](#)]
42. Oh, N.; Shirvani, P.P.; McCluskey, E.J. Control-flow checking by software signatures. *IEEE Trans. Reliab. (TR)* **2002**, *51*, 111–122. [[CrossRef](#)]
43. Rehman, S.; Shafique, M.; Henkel, J. Instruction scheduling for reliability-aware compilation. In Proceedings of the ACM/EDAC/IEEE Design Automation Conference (DAC), San Francisco, CA, USA, 3–7 June 2012; pp. 1288–1296.
44. Shrivastava, A.; Rhisheekesan, A.; Jayapaul, R.; Wu, C.J. Quantitative analysis of control flow checking mechanisms for soft errors. In Proceedings of the ACM/EDAC/IEEE Design Automation Conference (DAC), San Francisco, CA, USA, 1–5 June 2014; pp. 1–6.
45. Feng, S.; Gupta, S.; Ansari, A.; Mahlke, S. Shoestring: Probabilistic soft error reliability on the cheap. *ACM Sigarch Comput. Archit. News* **2010**, *38*, 385–396. [[CrossRef](#)]
46. Khudia, D.S.; Wright, G.; Mahlke, S. Efficient soft error protection for commodity embedded microprocessors using profile information. *ACM Sigplan Not.* **2012**, *47*, 99–108. [[CrossRef](#)]
47. Reis, G.A.; Chang, J.; Vachharajani, N.; Rangan, R.; August, D.I. SWIFT: Software implemented fault tolerance. In Proceedings of the IEEE International Symposium on Code Generation and Optimization (CGO), San Jose, CA, USA, 20–23 March 2005.
48. Yang, N.; Wang, Y. Identify silent data corruption vulnerable instructions using SVM. *IEEE Access* **2019**, *7*, 40210–40219. [[CrossRef](#)]
49. Laguna, I.; Schulz, M.; Richards, D.F.; Calhoun, J.; Olson, L. IPAS: Intelligent protection against silent output corruption in scientific applications. In Proceedings of the IEEE International Symposium on Code Generation and Optimization (CGO), Barcelona, Spain, 12–18 March 2016, pp. 227–238.
50. 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](#)]
51. Wang, N.J.; Quek, J.; Rafacz, T.M. Characterizing the effects of transient faults on a high-performance processor pipeline. In Proceedings of the International Conference on Dependable Systems and Networks, Florence, Italy, 8 June–1 July 2004; p. 61.