Low Cost Transient Fault Protection Using Loop Output Prediction

Sunghyun Park, Shikai Li, Scott Mahlke, *Fellow, IEEE*

Abstract—Performance-oriented optimizations have been successful by making transistor faster and smaller. However, the computer systems are becoming less reliable as optimizations increase their susceptibility to transient faults by reducing various design margins. The device cannot handle electromagnetic noise properly and naturally, the transient fault may lead to system failures or data corruptions. Given that transient fault occurs randomly in both time and space, the protection strategy should be constantly activated. Thus, the protection strategy should be cost-efficient and lightweight to be practical. In this paper, we propose *RSkip* which is a lightweight software-only protection technique. It focuses on minimizing the number of dynamic instructions for the fault protection. Rather than re-executing expensive identical computations, the output of re-computation is approximated and compared to validate execution. When the actual computation and estimation agree within predefined error bound, the computation is assumed fault-free and the expensive re-computation can be skipped. Prior instruction duplication work shows $2.89\times$ normalized execution time compared to unreliable execution over five compute-intensive benchmarks. With negligible loss of protection rate, *RSkip* reduces the overhead to $1.20\times$ by skipping 83.91% of re-computations.

Index Terms—Approximate Computing, Software-based Fault Protection, Value Prediction

1 Introduction

Over the past decades, researchers have been focusing on enhancing performance by reducing design margins, such as noise margins. However, the technologies make devices less reliable since the hardware with tight noise margins cannot handle electromagnetic noise properly. To protect against such faults, diverse techniques introduce redundancy at different levels ranging from hardware to software. In general, hardware techniques are effective but inevitably require the expensive design modification. Thus, researchers have proposed techniques exploiting redundant multithreading [1], [2], [3], [4] or instruction duplication under the guidance of the compiler [5], [6], [7], [8]. In particular, SWIFT [6] (SWIFT-R [7] with recovery), proposes software-only protection methodology and provides inspiration to recent explorations [8], [9]. However, the slowdown caused by increased dynamic instructions remains a concern which needs to be addressed.

In this paper, we focus on minimizing the runtime instruction overhead of instruction duplication-based protection strategy. We postulate that redundant instructions for protection techniques (referred to as *re-computation*) can be bypassed if the computation result can be predicted correctly. Otherwise, the re-computation must be executed to investigate a possible transient fault. Unlike traditional approximate computing techniques [10], [11], [12], mispredictions only cause run-time overhead without producing incorrect output since the prediction is used only for validation. To design an accurate and lightweight predict function, we exploit output similarity from output elements in spatial locality that are common in many applications. Based on output similarity, *dynamic interpolation* is proposed to estimate a computation result for a data element by capturing a local trend at runtime. By substituting the expensive re-computation with a linear equation, the large benefit can be achieved.

We propose *RSkip*, a static compilation system based on LLVM [13]. The system accepts the source code for unreliable program and creates a lightweight and resilient executable with full protection (fault detection and recovery). It is a fully automatic process without any hardware modification or additional work from the programmer. Runtime management is also inserted to deal with input variance. Throughout the paper, a *Single Event Upset* (SEU) fault model, which assumes an exact one-bit flip during the entire execution, is adopted as the fault model. Also, given that the commercial DRAMs are already protected by error correction codes (ECC), memory is assumed to be safe from the transient fault.

The major contributions of this work are as follows:

- We propose *RSkip*, a fully automatic compilation infrastructure, that provides lightweight and reliable execution. Over five compute-intensive benchmarks, RSkip shows $1.20\times$ slowdown compared to the unreliable execution. In contrast, SWIFT-R presents $2.89\times$ slowdown.
- We extend the applicability of software approximate computing techniques. Beside performance optimizations for approximable applications, they can be effectively adapted to detect faults in a cost effective and accurate manner.
- We introduce dynamic interpolation of loop output values to detect transient faults in lieu of re-computation. On average, 83.91% of re-computation in our target loops are skipped.
- The runtime management is proposed to automatically determine approximation aggressiveness towards diverse inputs.

2 Motivation

SWIFT(-R) duplicates instructions for fault detection (recovery requires triplication). At a synchronization point, SWIFT-R validates computed values to identify a possible transient fault. Synchronization points include *store*, *branch* and possibly *function calls* depending on the approaches. Once a fault is detected, a recovery mechanism will be triggered. SWIFT-R, SWIFT with a recovery mechanism, simply manages an additional copy of the program to conduct majority voting at synchronization points (similar to TMR). In general, a recovery mechanism can be studied independently. Because of the additional instructions for protection, SWIFT(-R) often suffers from high runtime overhead that may not be feasible in many situations (e.g., mobile device). Thus, we focus on reducing instruction overhead. SWIFT(-R) [6], [7] and RSkip has distinct approaches toward managing redundant copies of instructions. Rather than having extra identical instruction sequences, RSkip estimates computation results by using a simple prediction approach (e.g., regression model). If the approximate and original computation are similar, the output is considered correct and the SWIFT re-computation is bypassed. Otherwise, the re-computation will be performed to check for possible existence of a fault. **Note that mispredictions (false positives) result in runtime overhead without having direct impact on the program output. Missed faults can occur with false negatives, but the error is explicitly observed to be small, so such faults are unlikely to damage output quality noticeably.** When the values are not suitable for approximation, such as a pointer for address calculation or an induction variable, they are protected with traditional instruction duplication. Given their low computational overhead, their protection overhead remains marginal.

To design lightweight and accurate estimation model, we note *output similarity*. In general, *spatio-value similarity* [14] arises when data elements with spatial locality tend to be inherently consistent. Along with *spatio-value similarity*, we observe the data elements with spatial locality tend to share a certain trend. If trends can be captured efficiently, output values can be estimated more accurately. Based on this idea, *dynamic interpolation* is proposed to capture varying trends and react to them at runtime. The prediction overhead stays low by using a linear equation computed by two endpoints to estimate data points between them. The outliers will be considered as possible faults similar to perturbation screening [15] and examined more carefully. With dynamic interpolation, 83.91% of re-computations can be bypassed in the selected benchmarks.

3 System Overview

The overarching idea of RSkip is presented in Figure 1. RSkip takes an unreliable source codes as an input and automatically generates a lightweight resilient executable without any hardware modification, extra work from the programmer, and preprocessing on source codes or input data sets. Optionally, the programmer can specify as zero in specific code regions where 100% correctness is necessary. A default error bound will be applied when there is no annotation provided from the programmer. By

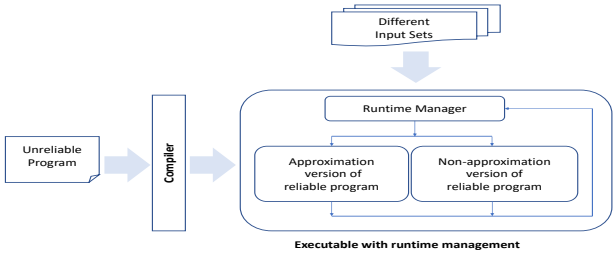


Fig. 1: System Overview

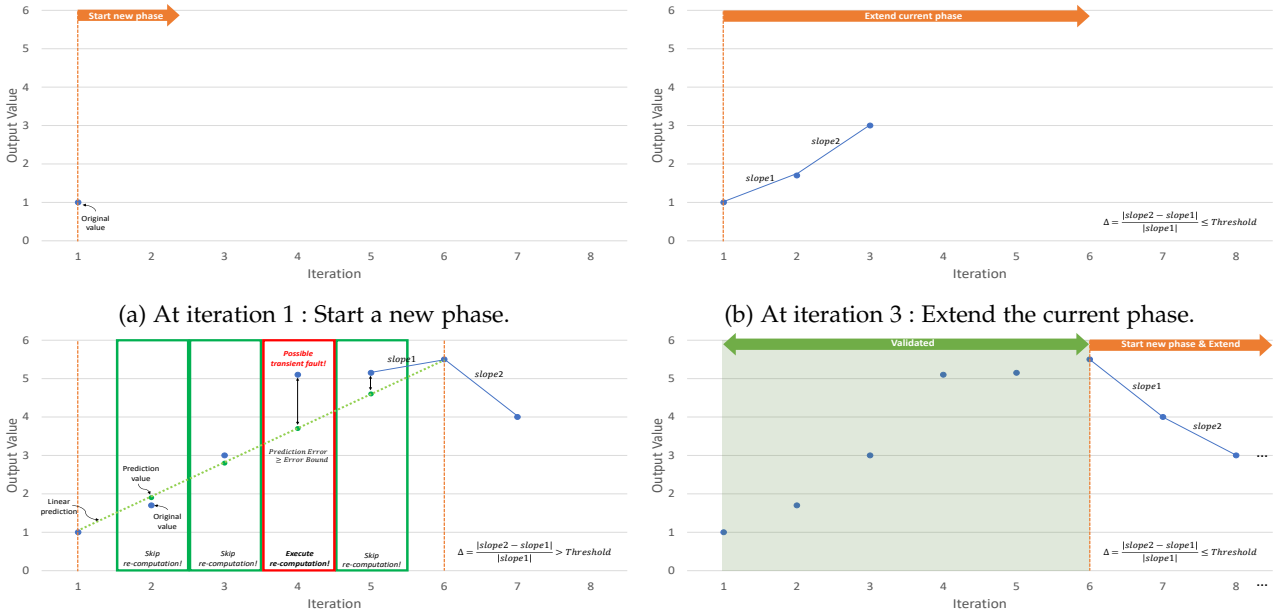
using static analysis on source codes, the compiler automatically detects optimization candidates containing certain patterns and transforms the code into the resilient form. Once the candidate is isolated, both approximate and non-approximate versions are created and one of two versions will be chosen at runtime. Non-approximation version of code is generated by conventional duplication technique. To react towards input diversity, *runtime manager* is also included in the executable and determines the version of the code. The non-approximation will be chosen in cases where the approximation techniques does not show any performance benefit. The codes, which are not identified as the candidate, will be transformed into non-approximation version without interaction with runtime management.

After one-time compilation process, the executable itself will observe the pattern of trends (referred to as *program context*). Program context can be quite different when the program runs with varying input sets or the same code region is executed multiple times with different live-in values within a single run. During the offline training process, it starts learning the tuning parameter which represents approximation aggressiveness. At execution, runtime manager creates a *program signature* using the runtime information for each transformed code region. It represents program context and is used to tune approximation aggressiveness for each situation. Since program performs repeats identical computation on different input sets, a certain group of input sets is likely to show a similar pattern of local trends. To group input sets with alike trend patterns, we assume the inputs that show similar statistics in the subset of data elements tend to share similar program context. Given that dynamic interpolation cuts phase dynamically based on recent slope changes, the program context is calculated with the statistics of slope changes. Also, two inputs might present similar trends only at the specific regions. To handle such cases, the executable will create program signature periodically and may generate more than one signatures during a single execution. During offline training phase, the best tuning parameter for each created signature is saved in a table. Later, on testing, runtime manager will create program signature and reference the table by using the signature to adjust its approximation aggressiveness during execution.

4 Dynamic Interpolation

4.1 Target pattern

In this section, we explain the implementations of approximation technique for the value prediction. Given the significant amount of time that compute-intensive benchmarks spend on loops, large loops are selected as main optimization target. Furthermore, SWIFT(-R) often struggles at loops



(a) At iteration 1 : Start a new phase.

(b) At iteration 3 : Extend the current phase.

(c) At iteration 7 : Cut the current phase.

(d) After iteration 7 : Repeat (b) - (c)

Fig. 2: The dynamic interpolation algorithm

due to repeating synchronization points which result in increase of dynamic instructions with dependencies. Therefore, the optimization for large loops is expected to bring notable performance improvement. In this paper, the values for store instructions in large loops will be predicted by employing dynamic interpolation. To apply optimization, the compiler automatically searches for the preferable patterns by conducting a thorough static analysis and transforms them into the resilient version with the prediction model. Nonetheless, certain essential information, such as loop trip count, cannot be known at compilation time. Therefore, the runtime management is inserted into the executable to handle dynamic information properly. In addition, not all large loops can benefit from our optimization. Certain patterns of loops, like an initialization loop, already presents low computation overhead. They are not considered as optimization target, thus these will be filtered out by static computation cost estimation. For simplicity, the computation containing loop or expensive function call are chosen as our optimization target.

4.2 Implementation

The linear prediction made by interpolation is totally depend on its two endpoints. In this paper, the data elements between two endpoints are considered to be in a *phase* and the length of a phase is also defined as *stride*. In general, due to the total dependence on the input set, the phase length cannot be determined statically. Therefore, the dynamic interpolation algorithm needs to be intelligent enough to maximize stride on a long trend or lower stride on widely fluctuating values. For fault protection, multiple copies of a program are executed together inside a single run. We note an opportunity to utilize a copy of program as a runtime guidance to cut the phase of a redundant copy dynamically.

Figure 2 demonstrates the dynamic interpolation algorithm step by step. In this example, a program signature and threshold (tuning parameter) are assumed to be known for simplicity. Figure 2a shows the setup stage of a new phase. Once the value for the first iteration is computed, the algorithm proceeds to the next iterations to see if the data elements are still in the same phase. Figure 2b shows the extension stage. If a change in the latest two slopes is less than the threshold, the previous phase extends its stride. While the condition satisfies, the algorithm maintains current phase and move onto the next iterations. Up to this point, only original computations are performed without making any prediction. Once a slope change is above the threshold, the phase is defined with the previous iteration and a linear prediction form the two endpoints is made to validate each computation. The cut stage is described at Figure 2c. When the difference between prediction and original computation is less than error bound, the algorithm assumes fault-free and bypass the re-computation. Otherwise, re-computation is triggered for the exact validation. The extension stage and cut stage are repeated after iteration 7 as in Figure 2d. The setup stage is no longer necessary as the next phase starts at the end of current phase.

Benchmark	Description	Computation Type of Prediction Target (Location of detected loop)
conv1d	1D convolution (Signal processing, Machine learning)	A reduction loop (inside a outer loop)
conv2d	2D convolution (Signal processing, Machine learning)	Nested reduction loops with conditional statement (inside a outer loop)
lud	LU decomposition (Linear algebra)	A reduction loop with a varying trip count (inside a outer loop)
sgemm	General matrix-multiplication (Linear algebra)	A reduction loop (inside nested loops)
Kde	Kernel Density Estimation (Machine learning)	Nested reduction loops (inside a outer loop)

Fig. 3: The selected benchmarks. The impact of skipping re-computation can be imagined by provided computation type and location.

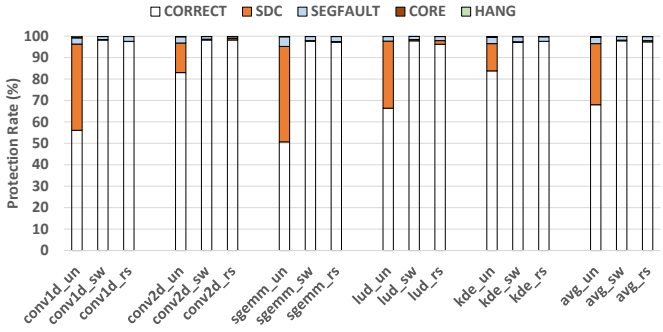


Fig. 4: The result of the fault injection experiment. Each protection scheme is tagged with the name of the application. (un : unprotected, sw : SWIFT-R, rs : RSkip)

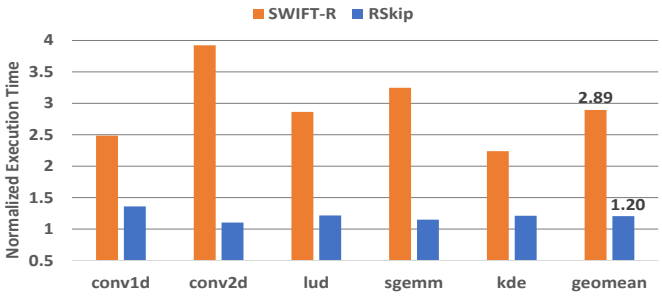


Fig. 5: Normalized execution time with test inputs of each benchmark

5 Evaluation

SWIFT-R [7] is chosen as the baseline for evaluation. Five compute-intensive applications with large loops are chosen as benchmarks from various domains. For detailed investigation, benchmarks are selected by considering diversity of detected patterns. The optimizations for the selected applications are important since most of them are heavily utilized kernels for real-life applications. The characteristics of each application are described in Figure 3. Both training inputs and test inputs are randomly generated. Throughout the experiments, a 20% error bound is assumed. The rationality of the error bound will be discussed in the following section.

5.1 Reliability

The fault protection rate of our work is evaluated using *Statistical Fault Injection* (SFI) [9], [16] in Gem5 [17] with out-of-order configuration of ARMv7-A. Each of five applications is executed 1,000 times with a randomized single bit flip on a register during each run. Due to the limitation of software-only technique, special registers, such as program counter, or stack pointer are excluded from the fault injection. Also, faults are limited to injections that fall within the detected loops to evaluate the reliability of RSkip. For analysis, the simulation result is categorized into five classes: "CORRECT", "SDC", "SEGFAULT", "CORE" and "HANG". If an injected fault results in corrupted output data, the execution is counted as "Silent Data Corruption(SDC)". If the program tries to access wrong memory address and cause the failure, it is counted as "SEGFAULT". It is regarded as "CORRECT" only when the execution generates the correct output without any error unlike the traditional approximation techniques that often allow a certain amount of error [9], [18]. However, in this paper, even small output errors are considered as unacceptable. Also, "CORE" and "HANG" represent the cases for core dump and hang, respectively.

Figure 4 shows the fault protection rate of each protection scheme. For comparison, the unprotected program is also included in the experimental results. On average, 68.00% of injected faults are masked on unprotected programs. And, they suffer from 28.5% SDCs and 3.08% SEGFAULTs. SWIFT-R shows a 97.82% protection rate, with the occurrences of SDC and SEGFAULT decreased to 0.32% and 1.66%, respectively. Such failures are caused by limitations of instruction duplication based techniques [6], [7]. With a 20% error bound, RSkip demonstrates a comparable level of protection rate to SWIFT-R by exhibiting a 97.34% protection rate. The occurrences of SDC and SEGFAULT are also reduced to 0.56% and 1.86%, respectively. Because of fuzzy validation inherent with the approximation approach, the SDC ratio of R-Skip is slightly higher than SWIFT-R. But, the difference is negligible. As the protection rates of the two approaches are very close, we can successfully prevent the side effect of fuzzy validation with 20% of error bound. Throughout the experiment, the occurrence for CORE and HANG are measured less than 1%.

5.2 Performance Overhead

The performance of each protection scheme is measured by running the benchmark executable with each approach on an Intel Xeon CPU E31230 with 3.20GHz. It has four cores, each with 32KB of I-cache and D-cache. The execution is forced to use a single thread and the performance overhead is measured after an automated training session with training inputs.

Figure 5 shows the performance with test inputs for all benchmarks. The execution time of each protection scheme is normalized by execution time of the unprotected program. On average, 83.91% of re-computation can be substituted with value prediction. As a result, RSkip shows the normalized execution time of $1.20\times$ while SWIFT-R suffers from $2.89\times$ slowdown due to the repeating synchronization points. The performance of previous work in *conv2d* is degraded by a large amount as the value is calculated in nested loops with conditional statements inside them. In this case, the benefit of skipping re-computation is clear. By skipping 89.83% of re-computation, RSkip shows the best-averaged performance improvement among selected benchmarks.

6 Related Works

Transient fault mitigation techniques can be broadly classified based on the form of redundancies utilized : hardware [19], [20], [21], thread [1], [3], [22] and instruction [5], [6], [7], [8]. In general, hardware-based techniques are powerful but expensive while software-based techniques are cheap but suffer from runtime overhead.

Especially, Oh et al. [5] first proposed instruction duplication based protection that replicates all instructions including memory operations with additional validations.

However, the technique shows great slowdown due to duplicated memory operations. To improve performance for instruction duplication scheme, Reis et al. proposed SWIFT [6] that removed unnecessary memory redundancies by building system on the assumption that memory is protected by ECC and restricting synchronization points. However, SWIFT has weakness on the protection for branch, load and store operations. nZDC [8] was proposed to tackle down this problem. As SWIFT provides detection strategy without recovery mechanism, Reis et al. expanded SWIFT further with TMR-based instruction level recovery technique [7]. Diverse recovery techniques [23], [24] are also explored. These recovery schemes can be integrated with RSkip if applicable. To reduce the performance overhead of traditional duplication scheme, the strategies injecting less instructions for protection are explored [9], [18]. In contrast to our work, they narrow down targets rather than utilizing approximate computing techniques. By introducing this concept, a better performance can be achieved with RSkip.

7 Conclusion

Transient fault force protection strategies should be fast and cost-efficient to be practical. In general, hardware-based techniques are too expensive due to inevitable hardware modification while software-based techniques suffer from runtime overhead due to increased dynamic instructions. RSkip demonstrated that the performance of the software-only protection strategy can be significantly improved by minimizing the number of additional instructions for protection scheme. By substituting the redundant computation with cheap estimation, the expensive re-computation can be bypassed. To design cost-efficient prediction model, the opportunity of utilizing the runtime information is explored. Furthermore, a runtime management is integrated to handle input diversity. As a result, the remarkable performance improvement is achieved in five benchmark applications. While SWIFT-R suffers from $2.89\times$ slowdown compared to the unreliable execution, RSkip only suffers $1.20\times$ slowdown by skipping 83.91% of expensive re-computation.

8 Acknowledgement

This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research (ASCR), under Award Number DE-SC0014134.

References

[1] Y. Zhang, J. W. Lee, N. P. Johnson, and D. I. August, "Daft: decoupled acyclic fault tolerance," *International Journal of Parallel Programming*, vol. 40, no. 1, pp. 118–140, 2012.

[2] Y. Zhang, S. Ghosh, J. Huang, J. W. Lee, S. A. Mahlke, and D. I. August, "Runtime asynchronous fault tolerance via speculation," in *Proceedings of the Tenth International Symposium on Code Generation and Optimization*. ACM, 2012, pp. 145–154.

[3] S. S. Mukherjee, M. Kontz, and S. K. Reinhardt, "Detailed design and evaluation of redundant multi-threading alternatives," in *Computer Architecture, 2002. Proceedings. 29th Annual International Symposium on*. IEEE, 2002, pp. 99–110.

[4] S. K. Reinhardt and S. S. Mukherjee, *Transient fault detection via simultaneous multithreading*. ACM, 2000, vol. 28, no. 2.

[5] N. Oh, P. P. Shirvani, and E. J. McCluskey, "Error detection by duplicated instructions in super-scalar processors," *IEEE Transactions on Reliability*, vol. 51, no. 1, pp. 63–75, 2002.

[6] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August, "Swift: Software implemented fault tolerance," in *Proceedings of the international symposium on Code generation and optimization*. IEEE Computer Society, 2005, pp. 243–254.

[7] G. A. Reis, J. Chang, and D. I. August, "Automatic instruction-level software-only recovery," *IEEE micro*, vol. 27, no. 1, 2007.

[8] M. Didehban and A. Shrivastava, "nzdc: A compiler technique for near zero silent data corruption," in *Proceedings of the 53rd Annual Design Automation Conference*. ACM, 2016, p. 48.

[9] D. S. Khudia and S. Mahlke, "Harnessing soft computations for low-budget fault tolerance," in *Microarchitecture (MICRO), 2014 47th Annual IEEE/ACM International Symposium on*. IEEE, 2014, pp. 319–330.

[10] A. Agarwal, M. Rinard, S. Sidiroglou, S. Misailovic, and H. Hoffmann, "Using code perforation to improve performance, reduce energy consumption, and respond to failures," Technical report, MIT, Tech. Rep., 2009.

[11] S. Sidiroglou-Douskos, S. Misailovic, H. Hoffmann, and M. Rinard, "Managing performance vs. accuracy trade-offs with loop perforation," in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. ACM, 2011, pp. 124–134.

[12] M. Samadi, D. A. Jamshidi, J. Lee, and S. Mahlke, "Paraprox: Pattern-based approximation for data parallel applications," in *ACM SIGARCH Computer Architecture News*, vol. 42, no. 1. ACM, 2014, pp. 35–50.

[13] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*. IEEE Computer Society, 2004, p. 75.

[14] J. San Miguel, J. Albericio, N. E. Jerger, and A. Jaleel, "The bunker cache for spatio-value approximation," in *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*. IEEE, 2016, pp. 1–12.

[15] P. Racunas, K. Constantinides, S. Manne, and S. S. Mukherjee, "Perturbation-based fault screening," in *High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on*. IEEE, 2007, pp. 169–180.

[16] S. Feng, S. Gupta, A. Ansari, and S. Mahlke, "Shoestring: probabilistic soft error reliability on the cheap," in *ACM SIGARCH Computer Architecture News*, vol. 38, no. 1. ACM, 2010, pp. 385–396.

[17] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti *et al.*, "The gem5 simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, 2011.

[18] R. Venkatagiri, A. Mahmoud, S. K. S. Hari, and S. V. Adve, "Approxilyzer: Towards a systematic framework for instruction-level approximate computing and its application to hardware resiliency," in *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*. IEEE, 2016, pp. 1–14.

[19] T. M. Austin, "Diva: A reliable substrate for deep submicron microarchitecture design," in *Microarchitecture, 1999. MICRO-32. Proceedings. 32nd Annual International Symposium on*. IEEE, 1999, pp. 196–207.

[20] C. Weaver and T. Austin, "A fault tolerant approach to microprocessor design," in *Dependable Systems and Networks, 2001. DSN 2001. International Conference on*. IEEE, 2001, pp. 411–420.

[21] Y. C. Yeh, "Triple-triple redundant 777 primary flight computer," in *Aerospace Applications Conference, 1996. Proceedings., 1996 IEEE*, vol. 1. IEEE, 1996, pp. 293–307.

[22] N. R. Saxena and E. J. McCluskey, "Dependable adaptive computing systems-the roar project," in *Systems, Man, and Cybernetics, 1998. 1998 IEEE International Conference on*, vol. 3. IEEE, 1998, pp. 2172–2177.

[23] S. Feng, S. Gupta, A. Ansari, S. A. Mahlke, and D. I. August, "Encore: low-cost, fine-grained transient fault recovery," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 2011, pp. 398–409.

[24] N. J. Wang and S. J. Patel, "Restore: Symptom-based soft error detection in microprocessors," *IEEE Transactions on Dependable and Secure Computing*, vol. 3, no. 3, pp. 188–201, 2006.