INVITED: Software Approaches for In-time Resilience

Aviral Shrivastava
Arizona State University
Aviral.Shrivastava@asu.edu

Moslem Didehban
Cadence Design Systems
Moslem@cadence.com

ABSTRACT

Advances in semiconductor technology have enabled unprecedented growth in safety-critical applications. However, due to unabated scaling, the unreliability of the underlying hardware is only getting worse. For a lot of applications, just recovering from errors is not enough – the latency between the occurrence of the fault to it's detection and recovery from the fault, i.e., in-time error resilience is of vital importance. This is especially true for real-time applications, where the timing of application events is a crucial part of the correctness of application. While software techniques for resilience are highly desirable since they can be flexibly applied, but achieving reliable, in-time software resilience is still an elusive goal. A new class of recent techniques have started to tackle this problem. This paper presents a succinct overview of existing software resilience techniques from the point-of-view of in-time resilience, and points out future challenges.

1 MOTIVATION

The unprecedented growth of software controlled systems has significantly increased the scope and the number of safety/mission critical applications – applications with unacceptable consequences of failure. Transient and permanent hardware faults pose a serious hardware reliability challenge for safety critical applications. While permanent faults or hard errors (e.g. stack at one/zero) have a lasting impact on microprocessor functionality, transient faults or soft errors (e.g. radiation-induced bit flip errors) do not cause any permanent damage to hardware and only tweak the value of some signal(s) or bit(s) impermanently. Although all hardware malfunctions will not lead to harm/loss, but even a small error can lead to catastrophic failures. It is predicted that overall system failure rate caused by hardware faults will continue to grow mainly due to ever-increasing level of integration (more transistors per core, more cores per chip and more chips per system)[1, 8, 12].

In many of the safety-critical applications, the time of application events is a crucial aspect of the correctness of execution. In such systems – that are commonly referred to as Cyber-Physical Systems (CPS) – in-time resilience is of vital importance. By in-time resilience, we imply that the time from the occurrence of a fault to the time that it is detected and recovered from, is minimized.

The prevalent assumption is that hardware and low-level error resilience solutions (e.g., the ARM Cortex-A76AE a Dual-core lock-step microprocessor) are more apt for achieving in-time, effective and efficient protection from hard and soft errors. This is because hardware techniques can trap most of the faults, they detect them fairly quickly, and the recovery is also pretty fast – just flush the pipeline, and start fetching again. At first, it may seem that such strategies incurs area overhead while software-level solutions (i.e executing computations twice on time and check the result) accompany considerable performance degradation. But in a multi-threading and multi-core world, area and space are essentially equivalent. So, n cores each with half performance (due to software redundancy) is same as $n/2$ cores (dual lock-stepped cores) with full performance. Furthermore, the throughput of a typical high-performance microprocessor is typically significantly more than a dual-core lock-stepped microprocessor.

While achieving resilience at lower level of design stack, i.e. circuit, microarchitectural and architectural level seems straightforward, such protection is rigid and cannot be customized based on high-level application requirements. Software solutions, on the other hand, can adjust their protection level based on application requirements and therefore achieve smart protection. Software-level resilience solutions can potentially provide resilience required for such applications without slowing down the execution of their performance-hungry portion or restricting their portability. This is especially useful for mixed criticality (in terms of error resilience) setup. As a result, this paper, we consider only software approaches for in-time resilience.

Since faults actually occur on hardware circuit and their impacts propagate to the software state, the scope of protection of a software resilience scheme is defined as a set of hardware components that the resilience transformation can prevent application failures due to faults on such components. Some techniques have a restricted scope-of-protection and they only protect one hardware components (i.e. register file or cache) and leave the rest components unprotected. In this work we focus on in-time software resilience schemes that their protection scope encompasses whole microprocessor core components excluding caches and memory hierarchy in their protection scope. The reason is that such components have already protected by error detection and correction codes in most modern microprocessors.

Ideally what we would like is to see software approaches that can provide in-time, effective and efficient resilience from hard and soft errors. Table 1 captures the most important related works in this domain. We organize the works into two categories: i) soft error protection schemes, and, ii) soft and hard errors protection solutions. Each of these categories is further subdivided into error detection schemes, and error recovery schemes. For each technique, we present its salient features and their limitations.

DAC '19, June 2–6, 2019, Las Vegas, NV, USA

ACM ISBN 978-1-4503-6725-7/19/06…$15.00
https://doi.org/10.1145/3316781.3323487

		Techniques	Description	Vulnerabilities
Soft Error Resilience	Error Detection (ED)	SWIFT[12], Shoestring[6]	• Duplicate arithmetic and logical operations • Check for errors in register operands of memory and control flow operations • Signature-based control-flow checking (only [12])	- Errors affecting memory operations - Valid-to-valid memory address change (only [6]) - Wrong-direction control flow errors - Short jump errors
		nZDC[4 , 5]	• Duplicate arithmetic and logical operations • Post store/CF error detection	- Errors on address part of silent stores - Short jump errors
	Error Recovery (ER)	SWIFTR[11], ELZAR[9]	• Triplicate arithmetic and logical operations • Voting between register operands of memory and control flow operations for ER	- Errors affecting memory operations - Wrong-direction control flow errors - Short jump errors - Unwanted memory write errors - Opcode-change errors (only[9])
		NEMESIS[3]	• Triplicate arithmetic and logical operations • Post store/CF error detection • On-demand voting for ER	- Unwanted memory write errors - Short jump errors
		Encore[7]	• Utilize read-only segments of code for efficient checkpoint/re-execution	- Wrong address memory writes errors - Unwanted memory write errors - Wrong-direction control flow errors - Short jump errors
		InCheck[2]	• Efficient and safe register file preservation • Post store/CF error detection • Memory and register file restoration for ER	- Errors on address part of silent stores - Unwanted memory write errors
Soft and Hard Error Resilience	Error Detection (ED)	SRMT[16]	• Duplicate application main thread • Send the source operands of memory operations from one thread to another for ED	- Errors affecting memory operations - Unwanted memory write errors
		Expert[14]	• Duplicate application main thread • Main thread updates the memory the other loads values from memory and check them for ED	- Errors on address part of silent stores
	Error Recovery (ER)	Fisher[15]	• Triplicate application main thread • Main thread updates the memory and the redundant threads perform disturbed ED • Thread state migration for ER	- Errors on address part of silent stores - Unwanted memory write errors

Figure 1: State of the art software level in-time resilience schemes.

2 IN-TIME SOFT ERROR RESILIENCE

An essential part of in-time error resilience is in-time error detection. Error detection can be implemented at different levels of granularity including, fine-grained (e.g. instruction level replication) and coarse-grained (e.g. thread level replication). The most basic coarse-grain scheme is program-level redundancy which in an application is executed two times and the redundant final results are used for error detection. Not only such solution cannot be used in the case of interactive or long-running applications, but they also cannot provide in-time error resilience which we define as the ability of a resilience solution for fast error detection/correction.

2.1 In-time Soft Error Detection Techniques

All existing fine-grained error detection schemes duplicate low-level assembly instructions and check the results of redundant instructions for soft error detection. The difference is in the type/number of replicated instructions and the position/number of checking operations. SWIFT[11] transformation duplicates the arithmetic and logical instructions of the program and inserts checking operations right before the execution of memory and control-flow instructions. SWIFT only executes one instance of memory and control flow operations and therefore if any soft error directly affect the execution of such instructions, the error remains undetected. For instance, transient faults hitting memory address generation unit while processing a load/store operations lead to access to an arbitrary memory location lead to failure in a the execution of a SWIFT-protected program. Similarly, errors directly affecting compare instructions register pointers in fetch or decode stage of pipeline can lead to an undetected wrong-direction control-flow errors i.e. a taken branch alters to not-taken or vice versa. Lastly, SWIFT transformation is vulnerable against short jumps errors and errors which cause program execution skips over the same number of main and redundant instructions within a basic block remain unnoticed.

Techniques after SWIFT mainly try to improve the SWIFT performance overhead by scarifying error detection capability. For instance, Shoestring[6] eliminates the need of instruction duplication for the chain of address computation instructions assuming

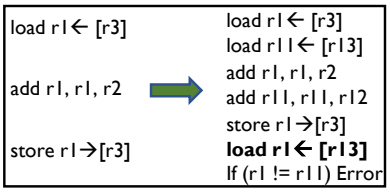

Figure 2: A read-modify-write code (left) and its corresponding nZDC transformation (right).

errors affecting them most probably cause a user-visible errors like segmentation faults. However, such transformation fails to detect the manifestation of errors which change the effective address of a memory (or the branch) operation to another valid address.

Prioritizing error detection capability over performance overhead, nZDC [4, 5] improves the error coverage of SWIFT transformation. The main contribution of nZDC is in protecting the instructions that were not duplicated by SWIFT-like schemes. nZDC introduces the idea of post critical instruction error checking. For example, to protect the store instruction, nZDC proposes a checking-load strategy as shown in figure2. Each memory write operation in a nZDC-protected program is followed by a memory read instruction from same memory location. Then error checking operations verify if the execution of store instruction has completed correctly. nZDC transformation also duplicates compare instructions, preserves their results in dedicated registers, and utilizes them for error detection purposes. nZDC micro-architectural level fault injection experiments on hardware components shows that eliminating memory and compare operation vulnerabilities significantly improves error resilience.

2.2 In-time Soft Error Detection+Recovery Techniques

Detection of faults is only half the story, and restarting (which is typically assumed when no recovery solution is provided) is not a solution for in-time error resilience. The goal of error recovery solutions is to detect and correct/mask the impact of errors from application output. For in-time error resilience, the most obvious recovery solution is forward-recovery-based. Forward error recovery solutions triplicate computations and perform majority voting operations to mask the impact of errors. SWIFTR[10] is a straightforward error recovery solution which triplicates arithmetic and logical computations and executes majority voting operations right before memory and control flow operations. More recent forward recovery solutions (like ELZAR[9]) utilizes vector and SIMD operations to hide the latency of triplicated instructions.

Nemesis[3] investigates the overall protection offered by simple forward recovery solutions. It claims that applications protected by such solutions encounter significantly more failures that their error detection predecessors due to three main facts: i) increased number of unprotected memory operations due to high register pressure, ii) substantial and frequent vulnerability window between software voting operations and the execution of memory or control flow operations, iii) wrong correction for already-propagated-to-memory errors. Note that ELZAR transformation is also vulnerable against all errors which may alter the opcode of a SIMD operations because in such situations all three redundant copies gets erroneous in the exact same way. Nemesis transformation improves SWIFTR shortcomings by performing on demand voting only after detecting a mismatch between the results of two versions of computations. Nemesis transformation skips over the execution of silent store instructions and therefore does not suffer from any silent store vulnerability.

It is more challenging to achieve in-time resilience using backward-recovery-based solutions. However, backward-recovery solutions may be preferred, since they require less resources. Backward recovery solutions only require executing two copies of the instructions, while forward recovery solutions require the execution of three copies of instructions. Backward recovery approaches create checkpoints – program architectural and memory state – and revert the application execution back to the last checkpoint in case of error. Main challenge here lies in making efficient (in terms of storage and performance) and error free checkpoints. Clearly preserving the whole program state incur significant storage and cost overhead which is not acceptable in most cases. To reduce the cost of checkpointing, EnCore[7] solution privileges the concept of idempotent region of codes – program regions that if program execution jumps back from any point of execution inside such regions to their beginning, program always produces correct results. In other words, idempotent region of codes are the region without conflicting read and write operations. Encore backward error recovery technique does not create explicit checkpoints and simply keeps track of idempotent regions of code and if error is detected inside of such regions, EnCore redirects the execution to the beginning of the idempotent region. Encore is based on the assumption of the existence of a perfect error detection scheme capable of detecting errors for the computation inside an idempotent region of code before they propagate to the outside of such regions. Even in that case, EnCore fails to provide recovery in several cases like control-flow errors, errors affecting address part of memory write operations and errors affecting non-idempotent region of codes.

InCheck[2] solution is a stand-alone fine-grained backward recovery technique which can accomplish in-time resilience with high level of error coverage. InCheck creates light-weighted and error free program architectural state checkpoints in the beginning of each program basic block and utilizes nZDC post-store error detection strategy. To make the overhead of application memory checkpointing acceptable, InCheck introduces the concept of single memory location checkpointing which materializes by inserting a memory backup read operation before the execution of each memory write operation. In the case of error, InCheck reverts the state of program to the beginning of basic block only if correct recovery is possible. Incheck vulnerabilities includes short-jumps errors and errors affecting address of a silent stores.

3 IN-TIME SOFT AND HARD ERROR RESILIENCE

Although instruction-level error resilience can provide flexible and effective error resilience, since they execute replicated computations on same hardware components they are unable to protect the execution against permanent hardware faults. To achieve protection against both transient and permanent faults by software level

transformations two main strategies exist: diverse computations and thread-level redundancy on a multicore microprocessor. Since the execution overhead of diverse computations is significantly more than thread-level redundancy schemes and they accomplish lower error coverage, we explore thread-level redundancy based solutions.

Note that the main presumption of thread-level redundancy for transient and permanent error resilience is that redundant threads can be executed on different cores of a multicore microprocessor. Such solutions privilege inherent hardware redundancy of multicore microprocessors, execute redundant computations on physically separate cores and check for the errors by comparing the results of redundant threads.

3.1 In-time Soft and Hard Error Detection Techniques

Expert[13] transformation is the sate-of-the-art thread-level scheme for hardware fault detection. It creates a redundant thread (called checker thread) for each application main thread and executes them in parallel on different cores. The execution of main and checker threads are mostly identical and they are required to be serialized for error detection on each memory write operation. Expert uses memory synchronization operations to guarantee that main-thread always executes store operations first and waits for checker thread to verify the results of write operations. Checker thread reads the recently updated memory locations and checks the data presented in the memory against the results of its own computation and raises error detection flag in the case of mismatch.

In Expert transformation checker thread never writes to the memory unless for synchronization purposes. Note that errors affecting the execution of checker-thread mainly remain inside the checker core and if propagate to the synchronization write operations can cause deadlock not SDC. Similar to nZDC and InCheck, Expert also in not capable of protecting computations against errors affecting the address of silent store instructions.

3.2 In-time Soft and Hard Error Detection+Recovery Techniques

FISHER[14] targets detection and recovery from both soft and hard error. FISHER executes three redundant threads on different cores and performs distributed and intertwined error detection and voting operations between redundant threads to provide error resilience and eliminate single-point-of-failures. Only one thread (named main thread) writes into the memory and both redundant threads independently verify the results of the main thread store operation by reading the store memory. Based on the results of these checks, FISHER transformation identify the erroneous core and perform thread state migration to revert the impact of error from computations. For instance, if both redundant checks fail, Fisher concludes that the core running main-thread is faulty and copy the state of one of redundant threads to the main thread. In addition, to make sure that the is disappeared (not permanent), FISHER double checks for errors in the computations after each error detection and recovery. FISHER transformation inherent silent store vulnerability from EXPERT and unwanted memory write errors from fine-grained soft error recovery solutions.

4 UNSOLVED CHALLENGES

In this work we reviewed the recent practices in the domain of software level in-time resilience for both transient and permanent errors. There are common vulnerabilities in each group of existing software-level in-time resilience. State of the art soft error detection solutions (e.g. SWIFT, Shoestring and nZDC) suffer from short jump errors and errors affecting the address of a silent store operation. In the later case, a write to a random memory location will get executed, however, since the store is silent the checking-load instruction receives the expected value from the memory and error remains undetected. For instance, if in snippet code shown in Figure2, `r2` register holds the value of Zero, the `add` instruction will not change the value of `r1` register and the store instruction writes an unmodified value back onto the memory. Therefore, even if store address gets faulty, the value of that checking-load instruction receives from the memory matches the value of `r11` register and no mismatch will be detected. All the forward and backward error recovery solutions suffer from unwanted-memory write errors. For instance, if due to a hardware fault, the opcode of an instruction changes to store, a random write to memory gets committed. When the impact of such error manifests itself as a discrepancy in one of the following voting or error checking operation, the impact of error can get reverted from operands (or the results of) critical instruction but not from the memory. In fact, there is no way for existing solutions to even realize if the execution has experienced an unwanted-write error.

REFERENCES

[1] Shekhar Borkar. 2005. Designing reliable systems from unreliable components: the challenges of transistor variability and degradation. *MICRO* (2005).

[2] Moslem Didehban et al. 2017. InCheck: An in-application recovery scheme for soft errors. In *DAC*. IEEE.

[3] Moslem Didehban et al. 2017. NEMESIS: A software approach for computing in presence of soft errors. In *ICCAD*. IEEE.

[4] Moslem Didehban and Aviral Shrivastava. 2016. nZDC: a compiler technique for near Zero Silent data Corruption. In *Proceedings of the 53rd Annual Design Automation Conference*. ACM, 48.

[5] Moslem Didehban and Aviral Shrivastava. 2018. A Compiler Technique for Processor-Wide Protection From Soft Errors in Multithreaded Environments. *IEEE Transactions on Reliability* 67, 1 (2018), 249–263.

[6] Shuguang Feng et al. 2010. Shoestring: probabilistic soft error reliability on the cheap. In *SIGARCH Computer Architecture News*, Vol. 38. ACM.

[7] Shuguang Feng et al. 2011. Encore: low-cost, fine-grained transient fault recovery. In *Proceedings of International Symposium on Microarchitecture*. ACM.

[8] Jörg Henkel, Lars Bauer, Nikil Dutt, Puneet Gupta, Sani Nassif, Muhammad Shafique, Mehdi Tahoori, and Norbert Wehn. 2013. Reliable on-chip systems in the nano-era: Lessons learnt and future trends. In *Proceedings of the 50th Annual Design Automation Conference*. ACM, 99.

[9] Dmitrii Kuvaiskii, Oleskii Oleksenko, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. 2016. Elzar: Triple modular redundancy using intel avx (practical experience report). In *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 646–653.

[10] George Reis et al. 2007. Automatic instruction-level software-only recovery. *IEEE micro* 27 (2007).

[11] George A Reis et al. 2005. Software-controlled fault tolerance. *TACO* 2 (2005).

[12] Muhammad Shafique, Siddharth Garg, Jörg Henkel, and Diana Marculescu. 2014. The EDA challenges in the dark silicon era: Temperature, reliability, and variability perspectives. In *Proceedings of the 51st Annual Design Automation Conference*. ACM, 1–6.

[13] Hwisoo So et al. 2018. EXPERT: Effective and flexible error protection by redundant multithreading. In *Design, Automation & Test in Europe Conference & Exhibition*. IEEE, 533–538.

[14] Hwisoo So et al. 2019. A software-level Redundant MultiThreading for Soft/Hard Error Detection and Recovery. In *Design, Automation & Test in Europe Conference & Exhibition*. IEEE.