Soft Error Resilience at Near-Zero Cost

Jianping Zeng
Purdue University
West Lafayette, IN, USA
zeng207@purdue.edu

Shao-Yu Huang
Purdue University
West Lafayette, IN, USA
huan1464@purdue.edu

Jiuyang Liu
Huazhong University of Science and Technology
Wuhan, China
liu@jiuyang.me

Changhee Jung
Purdue University
West Lafayette, IN, USA
chjung@purdue.edu

ABSTRACT

Among existing schemes for soft error resilience, acoustic-sensor-based detection stands out owing to its ability to prevent silent data corruption at low hardware cost. However, the state-of-the-art work not only incurs a considerable run-time overhead but also complicates the processor pipeline with intrusive microarchitectural modifications, hindering its practical deployment in real silicon. To this end, this paper presents VeriPipe, a near-zero-cost compiler/architecture codesign scheme for soft error resilience. VeriPipe compiler partitions input program to a series of regions (epochs) statically, while VeriPipe hardware verifies if they are error-free dynamically. In particular, VeriPipe achieves a simple yet efficient region-level verification where each region goes through a three-stage (*Execute*, *Verify*, and *Commit*) verification pipeline to ensure the absence of soft errors before proceeding to the next region. In particular, VeriPipe hardware overlaps the *Verify* stage of each region with the *Execute* stage of the next region, thereby effectively hiding the *Verify* delay. Experiments with 43 applications from SPEC2006/2017/NPB-CPP/SPLASH3/DoE Mini-Apps highlight the negligible overheads of VeriPipe, *i.e.,* an average of 1% run-time overhead and a storage overhead of only *3 registers* and *1 countdown timer*.

CCS CONCEPTS

• **Computer systems organization → Processors and memory architectures**; **Reliability**.

KEYWORDS

soft error resilience, compiler, computer architecture.

ACM Reference Format:
Jianping Zeng, Shao-Yu Huang, Jiuyang Liu, and Changhee Jung. 2024. Soft Error Resilience at Near-Zero Cost. In *Proceedings of the 38th ACM International Conference on Supercomputing (ICS '24), June 04–07, 2024, Kyoto, Japan.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3650200.3656605

1 INTRODUCTION

Soft errors have been the root cause of many real-world failures, especially for the large-scale high-performance computing (HPC) systems and datacenters [2, 7, 52]. In general, soft errors—also known as transient faults—are predominantly caused by the striking of high-energy particles[1], *e.g.,* cosmic ray and alpha particle from packaging materials, resulting in program crash or even worse silent data corruption (SDC) [21]. Soft error resilience becomes more important in the era of post-Moore's Law where near-threshold computing (NTC) plays a critical role in improving energy efficiency [30]. However, NTC makes the systems more vulnerable to soft errors, increasing their rate up to 30x higher than that of those systems with nominal voltage [30]. Thus, effective yet lightweight methods for detecting and mitigating soft errors are absolutely necessary not only to ensure program correctness but also to realize the full potential of NTC.

This necessity has sparked interest in innovative detection techniques. Among them, acoustic-sensor-based detection [61] is particularly prominent as the acoustic sensors eliminate silent data corruption (SDC); they directly sense the sound wave, which is always produced by particle striking as a physical phenomenon, thereby guaranteeing the full detection of soft errors, *i.e.,* none of them is missed. More importantly, the acoustic-sensor-based detection incurs minimal hardware costs [61]; only 0.0001% areal cost of the chip area (0.7mm^2; see Section 8) needs to be paid to deploy 30 sensors without requiring an extra metal layer for their interconnection.

Given the guaranteed error detection capability of acoustic sensors, researchers have leveraged them to realize soft error verification with the sensing latency in mind [43, 61, 69, 71]. The idea is that program execution prior to a given time T can be verified to be error-free the worst-case-detection-latency (WCDL) cycles later, provided that no sensor alarms in between. This makes senses because every soft error is to be detected within the WCDL cycles after its occurrence. In light of this, Liu *et al.* [43] proposed Turnstile to achieve core-level error containment. It enforces that data to be written must be error-free when they leave the core—with caches and memory protected by error-correcting code (ECC)—for no error to escape from the core.

To contain soft errors in the core, Turnstile delays writing the data of retired stores back to the L1 data cache, until they are verified to be error-free. That is, Turnstile leverages the store queue (SQ)[2] of each core as a redo buffer [26] to hold the retired stores for at least WCDL cycles till their data turn out to be verified. To avoid the SQ overflow during region execution, which would otherwise lead to incorrect error recovery, Turnstile compiler partitions program into a series of regions—comprising a sequence of instructions possibly including branches. As such, the stores of each region are verified as a whole once WCDL cycles are passed since the end of the region.

[1]Because of this, VeriPipe targets only such radiation-induced errors which we refer to as soft errors hereafter for simplicity.
[2]We assume a unified store queue in out-of-order cores without differentiating store buffer from store queue.

ICS '24, June 04–07, 2024, Kyoto, Japan

ACM ISBN 979-8-4007-0610-3/24/06
https://doi.org/10.1145/3650200.3656605

This is so-called *region-level error verification.* Notably, Turnstile turns the verification of registers into the memory verification by inserting stores to checkpoint region's *live-out* [3] registers—used by some following regions as inputs—to the memory.

Unfortunately, Turnstile is not practically implementable for two reasons: (1) its microarchitectural modifications are intrusive pressuring out-of-order pipeline optimization at design time; (2) it incurs a non-trivial run-time overhead, *i.e.,* 9% on average and up to 53%. To unveil why Turnstile leads to the high hardware complexity and the significant performance degradation, VeriPipe presents a new viewpoint of the *region-level error verification.* In Turnstile, all regions go through a three-stage (*Execute*, *Verify*, and *Commit*) verification pipeline. Each region begins with the *Execute* stage and transits to the *Verify* stage at the end of the region. Spending WCDL cycles thereafter (more precisely if no error is detected in the *Verify* stage for the WCDL cycles), the region finally moves to the *Commit* stage finishing the region verification. Upon the *Commit* of each region, Turnstile signals the store queue (SQ) to release the region's stores to the L1 data cache. With the help of this verification pipeline, Turnstile could overlap the *Verify* of an old region (*i.e.,* executed but unverified) with the *Execute* of a younger region, thus hiding the verification latency and achieving instruction-level parallelism (ILP).

Nonetheless, the *Execute* of a younger region is not always long enough to fully cover the *Verify* of the prior region, which causes the CPU pipeline to stall degrading the performance (see Section 2). Moreover, the *Verify* latency increases as WCDL goes up, *i.e.,* the CPU pipeline stalls more frequently, and each stall takes longer. In case the *Verify* delay cannot be fully hidden by a single region's *Execute* latency, Turnstile introduces a special hardware FIFO queue called region boundary buffer (RBB) that schedules multiple following younger regions for their execution time to overlap with the *Verify* of the oldest unverified region sitting at the RBB head.

Apart from the chip area occupancy, Turnstile's RBB puts significant pressure on realizing high-performance out-of-order pipeline—whose timing is already highly optimized—with the related control/signal and the critical path extension due to RBB-overflows stalling the pipeline. The crux of the problem is that this design challenge eventually prevents Turnstile from being fabricated on top of real silicon. In addition, Turnstile's register checkpoints (essentially stores), inserted for turning register verification into memory verification, are sometimes too many, thus incurring a non-negligible run-time overhead.

Thanks to our 3-stage pipeline modeling of Turnstile, we found out that its verification hardware can be dramatically simplified to only three registers and one countdown timer. In fact, only one region on the *Execute* stage is enough—if it is sufficiently long—to fully cover the latency of the prior region's *Verify* stage. With the above finding in mind, VeriPipe proposes a simple yet efficient verification pipeline where the *Verify* latency of a region can be hidden by only one following region execution, which obviates the need of complex RBB-like hardware. The key idea is to let each region get verified at the end of the next region. Figure 1 shows how VeriPipe's simplified three-stage verification pipeline works. As usual, each region starts with the *Execute* and moves to the *Verify* when reaching its end; however, the region here reaches the *Commit* as soon as the next region finishes with no error detected.

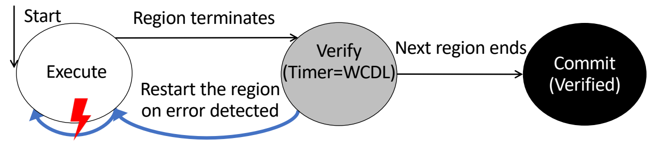

Figure 1: VeriPipe's region verification automaton

However, it is challenging to ensure that the execution time of each region is never smaller than WCDL cycles. To overcome this challenge, VeriPipe proposes a simple yet effective hardware technique called *region stitching*. At run time, it combines any short region (whose execution time is less than WCDL) with the following regions so that the stitched region's execution cycles are at least WCDL. This allows VeriPipe to fully hide the *Verify* latency of every region! *The beauty of region stitching is that it scales to arbitrarily long WCDL with neither storage overhead—other than only one logic gate for control—nor run-time overhead.* In addition, VeriPipe compiler eliminates redundant checkpoint stores, lowering the run-time overhead further without jeopardizing the soft error resilience guarantee.

The experiments with 43 applications from SPEC2006/2017 [9, 24], NPB-CPP [46], SPLASH3 [56], and DoE Mini-Apps [29, 60] demonstrate that VeriPipe incurs only a 1% run-time overhead on average regardless of WCDLs. In summary, VeriPipe:

- Incurs near-zero run-time overhead with the intelligent compiler/architecture codesign regardless of long WCDL.
- Incurs only a 0.018% areal cost according to the RTL synthesis results with TSMC 7nm technology, reducing Turnstile's hardware cost and power consumption by ≈91%.
- Is the first to achieve near-zero-hardware-cost soft error resilience, making its commercialization possible in silicon.

2 BACKGROUND AND MOTIVATION

2.1 Acoustic-Sensor-Based Soft Error Detection

Recently, Upasani *et al.* [61] proposed to detect soft errors using acoustic sensors. In the event of energetic particle striking (*e.g.,* cosmic ray and alpha particle), the sensors perceive the acoustic wave—generated by the physical phenomenon of the striking—and thus always detect the resulting soft error. According to the prior work [61], an acoustic sensor only occupies the same die size as one 6T SRAM bit, *i.e.,* $0.027\mu m^2$ with TSMC 7*nm* node [10].

Nevertheless, this detection scheme cannot immediately detect soft errors due to inherent sensing delays. Consequently, errors might bypass the detection and eventually propagate the corrupted data to ECC-protected memory, causing detected-but-uncorrectable errors (DUEs). To address this issue, the prior work [61] includes caches in the error containment domain. It holds L1 dirty cachelines for WCDL until they are verified to be error-free before their write-back to L2. Unfortunately, this design requires significant changes to the existing cache structures and their cache coherence protocol.

2.2 Region-Level Soft Error Verification

To tolerate the detection latency without changing caches, Liu *et al.* proposed Turnstile [43] that contains errors in the core. Turnstile holds data being stored in the SQ for WCDL before merging them to the L1 data cache. To avoid the SQ overflow that leads to incorrect error recovery, Turnstile compiler partitions input program into a series of verifiable/recoverable regions with SQ size in mind. More precisely, each region is formed so that the number of its stores never exceeds the half of SQ size; that way while a region is under verification occupying a half of the SQ, the other half can be used to accommodate the stores of the following region(s). For example, Figure 2 (a) and (b) shows that input program is divided into $Rg1$ and $Rg2$ such that each region has at most 2 stores since SQ size is 4 here. Once region boundaries are delineated, Turnstile compiler identifies each region's live-out registers and checkpoints them to the memory for recovery purpose, *e.g.,* checkpoint $1c$ and $2c$ that are essentially stores.

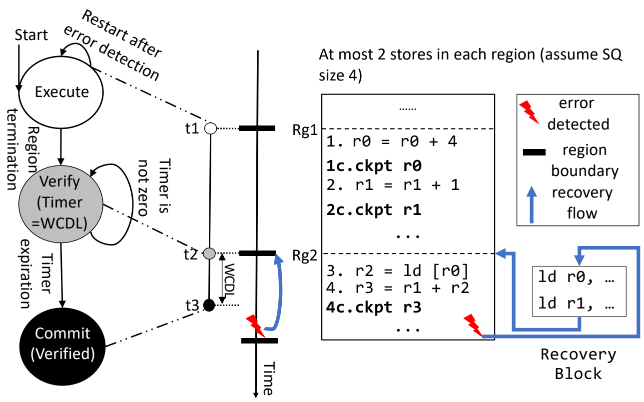

Figure 2: (a) Turnstile's region verification automaton; (b) store queue aware region partitioning; eager checkpointing

Turnstile hardware checks the integrity of the regions, *i.e.,* the absence of a soft error during their execution, by leveraging the verification pipeline as shown in Figure 2 (a). For example, $Rg1$ enters into the *Execute* at $t1$ and moves to the *Verify* as soon as it finishes execution at $t2$. Later, $Rg1$ reaches the *Commit* at $t3$ after spending WCDL cycles since its end, provided none of deployed acoustic sensors alarms the occurrence of soft errors in between.

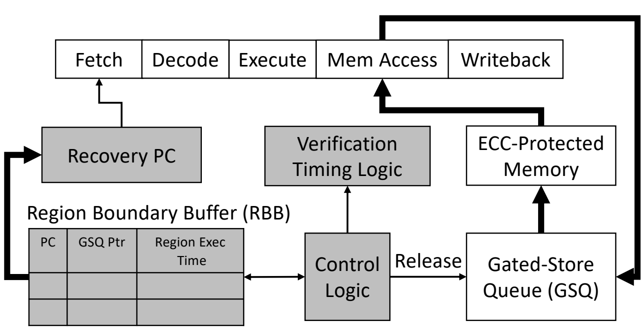

Figure 3: Turnstile architectural diagram with marking newly added components gray; bold lines correspond to data paths while thin lines to control paths

To implement the above verification pipeline, Turnstile proposes several hardware components—marked gray in Figure 3—with the existing store queue (SQ) repurposed as a gated store queue (GSQ). During the execution of each region, the GSQ holds the stores of the region until it is verified, though they are already retired from the reorder buffer (ROB). Subsequently, if no errors are detected within WCDL, the verified GSQ entries are to be merged to the L1 data cache. To figure this out, Turnstile introduces a FIFO queue called region boundary buffer (RBB) that tracks the progress of regions being executed or verified. When the ROB commits a region boundary, Turnstile allocates the corresponding entry in the RBB; the core pipeline stalls when encountering a region boundary if RBB is already full. Each RBB entry contains 3 items: (1) the PC of the next instruction, (2) GSQ Ptr—pointing to the tail of the GSQ—for releasing the stores of the region when it reaches the *Commit*, and (3) timing information for determining when the region verification completes, *i.e., Commit.* With the FIFO nature of RBB, Turnstile keeps the head of the RBB up-to-date with the oldest unverified region. For example, Figure 2 shows that $Rg1$ becomes error-free at $t3$. As a result, Turnstile (1) signals the GSQ to write back $Rg1$'s stores (*e.g.,* $1c$ and $2c$) to the L1 data cache, (2) copies the PC field of the head RBB entry to *Recovery PC* in case an error occurs thereafter, and (3) deallocates the head entry with it set to the next entry becoming the oldest among unverified regions.

Upon soft errors (⚡) detected, Turnstile performs three actions to recover from them: (1) discarding (unverified) GSQ entries which are younger than the *GSQ Pointer*, (2) executing the code in a recovery block—shown in Figure 2 (b)—to reload the oldest unverified region's live-in registers from a dedicated checkpoint storage in the memory, and (3) resuming the program execution from the region's beginning pointed to by *Recovery PC.*

2.3 Limitations of Prior Work

Unfortunately, Turnstile's RBB and its control logic require complex peripheral circuitry, resulting in way longer access latency than VeriPipe, *e.g.,* Turnstile ($0.26ns$) vs VeriPipe ($0.07ns$) as shown in Table 1. This implies that Turnstile restricts the core frequency to a maximum of 3.86GHz, making it impossible to be used for current and future high-end processors. Even if future process technologies might be ready for higher clock frequency, Turnstile's intricate peripheral circuitry poses a significant challenge in reducing its wire delay—scaling more slowly compared to transistor delay—as highlighted by prior work [1, 47] on processor core design. Consequently, it is a daunting challenge to increase the clock frequency of Turnstile-enabled processors in the future.

Another prior work Turnpike [69] also leverages GSQ to contain soft errors in an in-order that usually has a tiny SQ, *e.g.,* 4 SQ entries in ARM Cortex-A53 [33]. To lower the pressure on the tiny GSQ caused by store verification, Turnpike compiler eliminates unnecessary stores, while its hardware early releases certain stores to the L1 data cache without holding them in the GSQ for verification. However, Turnpike incurs a high run-time overhead for out-of-order cores, despite its additional hardware support for the fast store release. This is because Turnpike compiler fails to form large regions for out-of-order cores—as with Turnstile compiler—and thus quickly overflows RBB , causing the core pipeline to stall frequently. Moreover, Turnpike's fast store release turns out to be unnecessary for two reasons: (1) stores are off-the-critical-path in out-of-order cores thanks to their large SQs, *e.g.,* SQ size of ARM Cortex-A77 is 90 [38]; (2) their dynamic scheduling easily finds non-store instructions even if the SQ is full.

3 OVERVIEW OF VERIPIPE

What makes VeriPipe stand out from prior schemes [43, 69] is that it always verifies a region to be error-free at the end of the next region. As such, each stage of VeriPipe's *3-stage verification pipeline* is always occupied by at most one region (see Figure 1). This allows VeriPipe to track the region verification with minimal hardware cost, unlike the prior schemes that require expensive RBB whose overflow leads to significant core pipeline stalls.

To realize the low-cost *3-stage verification pipeline*, VeriPipe only introduces 3 registers, *e.g., GSQ Pointer*, *Region Register*, and *Recovery PC*, and 1 countdown timer, as shown in Figure 4. *Recovery PC* refers to the end of the latest verified region, *i.e.,* whenever *Commit* stage gets a new region, it is pointed to by *Recovery PC* to serve as a recovery point. *Region Register* is a pointer referring to the last instruction of the region on the *Verify*; this is technically a region boundary instruction and thus points to the beginning of the next region that is currently on the *Execute*. *GSQ Pointer* refers to the tail of gated store queue (GSQ), indicating that all the following younger stores are not verified yet and thus must be squashed in case of a soft error. Finally, to track the remaining cycles for a region on the *Verify* to transit to *Commit*, the timer is reset to WCDL cycles at each region boundary and counts down each cycle thereafter.

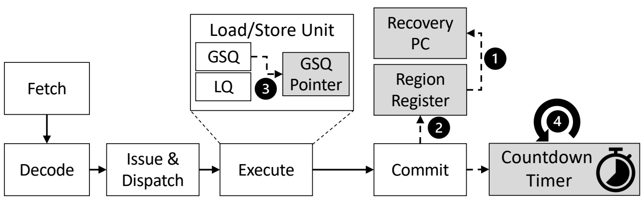

Figure 4: VeriPipe microarchitecture; shaded are newly added

When a region finishes its execution, *i.e.,* the region boundary instruction is committed from the core pipeline, the prior region—whose end has been pointed to by *Region Register*—moves on to the *Commit*. VeriPipe then updates its registers and countdown timer accordingly as shown in Figure 4: (❶) updating *Recovery PC* with the current *Region Register*, (❷) resetting it to the address of the region boundary instruction, (❸) releasing the stores older than *GSQ Pointer* to the L1 data cache with *GSQ Pointer* reset to the current tail of the GSQ, and (❹) resetting the countdown timer to WCDL cycles.

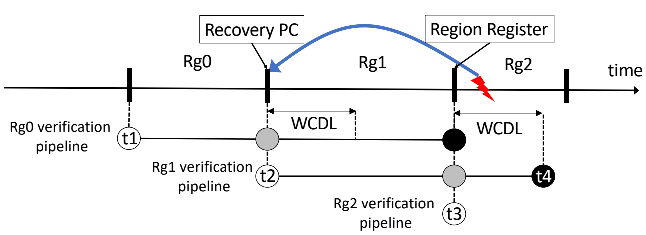

Figure 5: Verification pipeline status at time $t3$**;** $Rg0$ **has been verified to be error-free;** $Rg1$ **on** ***Verify*****, while** $Rg2$ **on** ***Execute***

Figure 5 shows how the *three-stage verification pipeline* works using the three registers and the countdown timer. Suppose all regions $Rg0$-$Rg2$ are long enough to cover WCDL cycles. At $t1$, $Rg0$ enters into the *Execute*, and VeriPipe resets the timer to WCDL cycles. At $t2$, $Rg0$ moves to the *Verify*, $Rg1$ starts with the *Execute*, and VeriPipe resets the timer again. While the core pipeline reaches the end of $Rg1$ at $t3$, $Rg0$ reaches the *Commit*. Accordingly, VeriPipe updates *Recovery PC* with the end of $Rg0$ and sets *Region Register* to the end of $Rg1$. After resetting the timer, VeriPipe starts to verify $Rg1$. Once a soft error (ϟ) is detected in $Rg2$, VeriPipe consults *Recovery PC* to resume the program execution from $Rg0$'s end.

4 VERIPIPE COMPILER

This section illustrates how VeriPipe compiler forms a series of verifiable/recoverable regions as shown in Figure 6.

4.1 Region Partitioning

VeriPipe—akin to prior work [69]—leverages the gated store queue (GSQ) as a redo buffer to hold the data being stored for verification. To circumvent GSQ overflow which would otherwise cause incorrect error recovery, VeriPipe compiler partitions input program into a series of regions with half of the GSQ size in mind. Thus, the GSQ never overflows when a region is being executed while a prior region is on the *Verify*. VeriPipe compiler first partitions program at callsites and loop headers. Specifically, it inserts a region boundary at all entry and exit points of functions. To avoid GSQ overflow during the execution of loops, VeriPipe compiler also inserts a region boundary in the loop headers, *i.e.,* each loop iteration forms a region. Starting with these initial boundaries, VeriPipe compiler counts the stores while traversing the control flow graph (CFG) of the input function; it picks the maximum of store counts from multiple paths at joint points. When the count reaches the threshold—*i.e.,* half of GSQ size, a region boundary is inserted and serves as a recovery point in case the following region gets interrupted by soft errors.

4.2 Live-Out Register Checkpointing

However, the GSQ only verifies memory data, leaving register values unverified. To address this issue, VeriPipe turns register verification into memory verification by checkpointing registers to the memory. First, VeriPipe compiler identifies *live-out* registers in each region using liveness analysis [3]; these registers are used as inputs of subsequent regions. Later, VeriPipe compiler checkpoints *live-out* registers of each region to the memory by inserting stores after their most recent definitions in the region; VeriPipe still ensures that none of regions has more stores than the partitioning threshold. In particular, a region's checkpoints will be used to recover a subsequent region in case of soft error detected. As such, soft errors occurred in a region do not compromise the region's error recovery since its recovery uses the checkpoints in the prior regions that must be already verified before proceeding to the error-interrupted region.

4.3 Checkpoint Elimination

VeriPipe's register checkpointing inserts checkpoint stores that incur more run-time overhead. VeriPipe exploits four existing compiler optimizations to eliminate unnecessary checkpoints while still maintaining the soft error resilience guarantee.

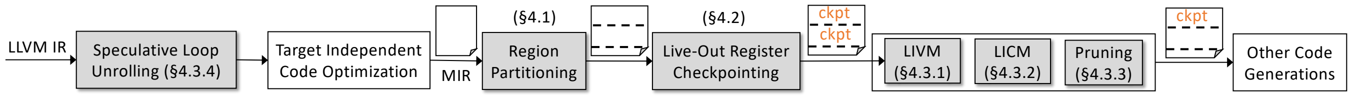

Figure 6: VeriPipe compiler workflow; shaded passes are newly proposed

4.3.1 Loop Induction Variable Merging (LIVM). Existing compilers use loop strength reduction (LSR) [3] to replace an expensive expression of induced induction variables, such as multiplication of computing array element's address, by a cheap addition of basic induction variables and constants. Figure 7 (b) shows that the calculation of an array element's address is replaced by the addition ($r1 = r1 + 4$) of a basic induction variable $r1$. However, LSR leads to more checkpoints as (1) it introduces more basic induction variables involving loop-carried dependence, *e.g.,* $r0$ and $r1$ in Figure 7(b); (2) they are live-out to the next loop iteration (*i.e.,* region) and thus must be checkpointed, *e.g.,* $5c$ and $6c$ in the figure.

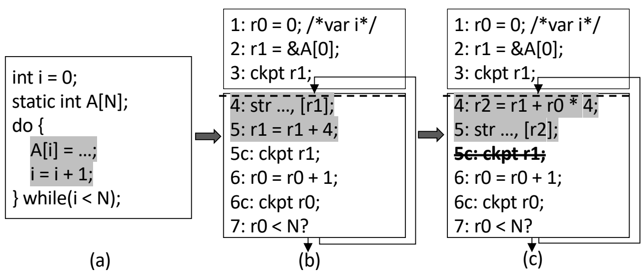

Figure 7: (a) original C code, (b) assembly code with LSR enabled, and (c) eliminating checkpoint $5c$ by LIVM

To eliminate the loop-carried dependencies for certain registers and their corresponding checkpoints, VeriPipe implements the same loop induction variable merging (LIVM) of prior work [69]. LIVM merges induced induction variables into the expressions of basic induction variables, resulting in only one checkpoint per induction variable chain. For example, Figure 7 (c) shows that $r2$ is now computed using basic induction variable $r0$ and constants. This eliminates the checkpoint $5c$. Consequently, LIVM brings a significant performance improvement for loop-intensive applications.

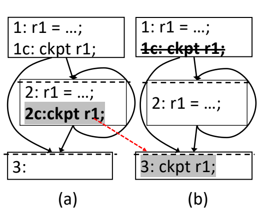

Figure 8: (a) original code, (b) moving checkpoint $2c$ out of loop with LICM

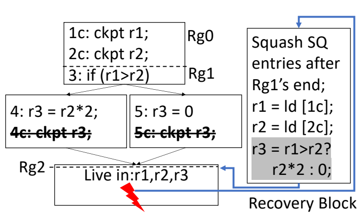

Figure 9: Eliminate checkpoint $4c$ and $5c$ with optimal pruning; recovery process on the right

4.3.2 Loop-Invariant Checkpoint Motion (LICM). For certain remaining checkpoints inside loops, VeriPipe employs the same loop-invariant checkpoint motion (LICM) as in Turnpike [69]. LICM can safely move checkpoints from within loops to their exit blocks, provided that the checkpointed registers are loop-invariant, *i.e.,* no write-after-read (WAR)-dependence on them in the loops. Figure 8 shows that checkpoint $2c$ is moved out of the loop to the bottom basic block because $r1$ is loop-invariant. Moreover, with $2c$ moved to the bottom basic block, $1c$ in the top basic block becomes redundant and thus can be eliminated, enabling synergy further.

4.3.3 Optimal Checkpoint Pruning. To further reduce the run-time overhead caused by checkpoints, VeriPipe exploits the optimal checkpoint pruning—invented by [32]—to eliminate redundant checkpoints since they can be reconstructed using constants and/or other remaining checkpoints at recovery time. For example, Figure 9 shows that checkpoint $4c$ and $5c$ are eliminated. In the wake of soft error interruption, VeriPipe runtime recomputes register $r3$'s value using the checkpoint $1c/2c$ and immediate values in the recovery block and resumes the program execution from the beginning of $Rg2$. Consequently, VeriPipe shifts checkpoint overhead from run time to the recovery time, significantly lowering the run-time overhead without jeopardizing the soft error resilience guarantee.

4.3.4 Speculative Loop Unrolling. Recall that VeriPipe compiler inserts a region boundary in all loop headers to avert GSQ overflow during loop execution. However, this often generates a lot of short regions given that small loops are prevalent in the evaluated benchmarks; Figure 10 shows that 50% of the loops in the evaluated applications have less than 30 instructions. Given that registers tend to be short-lived [48] and thus long regions likely have fewer live-out registers to be checkpointed, VeriPipe's region partitioning generates superfluous checkpoints for the short regions.

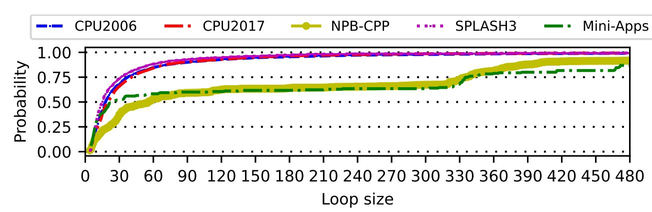

Figure 10: CDF of instruction count in loops

To address the above issue, VeriPipe applies the *speculative loop unrolling*—invented by Capri [28]—to enlarge loops no matter if their iteration counts are static-unknown; conventional loop unrolling only unrolls the loops with constant iteration counts[3]. While duplicating a loop body for a certain number of times, VeriPipe compiler inserts the code to check for proper loop termination after each duplicated loop body. To compute a proper unrolling factor, VeriPipe takes an optimistic approach that assumes each instruction finishes in one cycle on out-of-order cores with commit width CW.

[3]GCC can unroll some loops with static-unknown iteration counts if they are computed as expressions, while VeriPipe's unrolling is generic to any loops.

Thus, the unrolling factor is computed as $\frac{WCDL \times CW}{\text{Loop Size}}$ to balance code size with the size of unrolled loops. In particular, VeriPipe carefully tunes speculative loop unrolling to incur as little negative impact on the performance as possible.

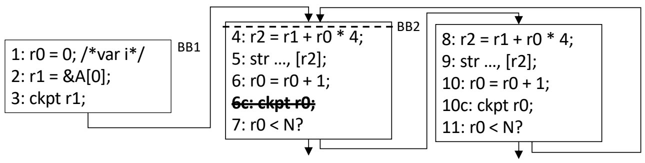

Figure 11: Reduce the dynamic instances of $r0$'s checkpoint by a factor of 1/2 via speculative loop unrolling

Once regions are enlarged, certain registers become no longer live-out anymore, rendering their checkpoints redundant. For example, suppose WCDL is 10 cycles while CW is 1 on an out-of-order core, Figure 11 shows that VeriPipe unrolls the loop in Figure 7(c) by a factor of 2 [45]. That way, checkpoint $6c$ becomes unnecessary and can be eliminated since $r0$ defined at line 6 is not live-out anymore. Consequently, VeriPipe reduces the number of dynamically executed checkpoints by 2x.

5 DEALING WITH SHORT REGIONS

Recall that VeriPipe compiler inserts a region boundary at all entry and exit points of functions, this generates a lot of short regions for small functions if they are recursive and called inside loops. Even worse, in certain evaluated applications, *e.g.,* `povray` and `leela`, many small recursive functions cannot be transformed to be non-recursive by tail call elimination [3] for inlining. Here, the problem is that the presence of short regions causes incorrect region verification. This is because at the end of a short region, WCDL cycles have not passed since the end of the prior region that is on the *Verify*. Therefore, moving the prior region to the *Commit* could release potentially corrupted data to the L1 data cache, resulting in detected-but-uncorrectable errors (DUEs).

5.1 Naive Dynamic Enforcement

To address the above issue, one naive way is artificially extending the execution time of the short region to be WCDL cycles. That is, the core pipeline stalls at the end of each short region until the countdown timer hits zero, *i.e.,* WCDL cycles have passed since the end of the prior region that is on the *Verify*. However, the naive approach incurs a significant run-time overhead due to frequent core pipeline stalls occurred at the end of short regions, which becomes even worse for longer WCDLs; Section 8.2 shows that this strategy incurs an average of 8% and up to 50% run-time overhead.

5.2 Region Stitching

We find out it is unnecessary to stall the core pipeline at the end of a short region as long as the stores of the prior region are held in the GSQ for WCDL cycles. In light of this observation, VeriPipe proposes a simple hardware technique called *region stitching* that dynamically combines a short region with the following regions while holding the stores of the prior region in the GSQ. VeriPipe continues this process until the countdown timer hits zero at a region boundary, *i.e.,* the stitched region is now long enough to cover WCDL. We can even relax this for higher performance. When the timer becomes zero, the prior region is immediately verified to be error-free without waiting for the next region boundary. This early releases the region's stores from the GSQ to the L1 data cache. *The beauty of region stitching is that it does not incur any storage overhead and scales up to arbitrarily long WCDL.*

```
1  Function VerificationController(T, I):
       Data: Countdown Timer T
       Data: Instruction I
2      if T == 0 then
3          foreach Str ∈ GSQ[start_index, GSQ Pointer] do
4              merge Str to L1 data cache;
5          end
6          if I is a region boundary instruction then
               /* Performing the following 4 steps
                  simultaneously at circuit level    */
7              T ← WCDL;
8              Recovery PC ← Region Register;
9              Region Register ← I's PC;
10             GSQ Pointer ← GSQ tail;
11         end
12     else if I is a region boundary instruction then
13         Treat I as a noop;
```

Algorithm 1: Verification Controller

Algorithm 1 describes VeriPipe's actions upon committing an instruction, which accepts two inputs: the countdown timer T and the instruction I. If T hits zero (at line 2), VeriPipe early merges the stores older than *GSQ Pointer* to the L1 data cache since they are already verified to be error-free. This relieves the pressure on the GSQ while still maintaining soft error resilience guarantee. If I is a region boundary instruction, VeriPipe updates its three registers accordingly and resets the timer as shown in the algorithm from line 7 to 10. Otherwise, VeriPipe treats the region boundary instruction as a noop, doing nothing special. In particular, region stitching still verifies inserted checkpoints, though it might render some checkpoints not live-out and thus unnecessary.

Most important, region stitching still works correctly for synchronization primitives, *e.g.,* atomic operations and memory fences, which force all prior stores to be merged into the L1 data cache before committing them. This is because VeriPipe treats the primitives as region boundaries during region formation (see Section 4.1). Therefore, stores prior to the synchronization primitives are held in the GSQ for verification until the regions ending at the primitives reach the *Commit*. After that, the stores of the regions are released to the L1 data cache, *i.e.,* they become visible to other cores, allowing the ROB to commit the primitives. Consequently, other cores competing for the primitives can start their execution without observing any unverified data.

6 RECOVERY PROTOCOL

To resume program from soft errors, VeriPipe performs three actions in a row: (1) discarding the stores younger than *GSQ Pointer* from the GSQ, (2) restoring live-in registers of the oldest unverified region in a compiler-generated per-region recovery block (see Figure 9), and (3) resuming the program execution from the region's beginning. Figure 12 shows how VeriPipe recovers the program with region stitching enabled. Assume *Rg*0-*Rg*1 and *Rg*5-*Rg*6 are long enough to cover WCDL, while *Rg*2 − *Rg*4 are not.

To start with, *Rg*0 is on the *Commit* (*i.e.,* it is error-free), and *Rg*1 is on the *Verify*, as shown in Figure 12 (a). Upon reaching the end of *Rg*2—on the *Execute*, VeriPipe combines *Rg*2, *Rg*3, and *Rg*4 into a single region referred to as *Rg*234 hereafter to ensure that the total execution time of *Rg*234 meets or exceeds the required WCDL cycles. When a soft error is detected in either *Rg*2 or *Rg*3, it is safe to resume the program from *Rg*1's beginning pointed to by *Recovery PC*. This is because *Rg*1's live-in registers are already verified to be error-free in prior regions; all stores in *Rg*1 − *Rg*4 are squashed from the GSQ and thus do not affect the memory states.

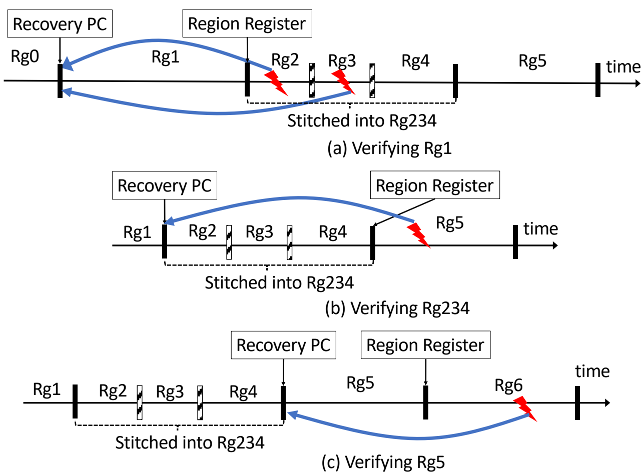

Figure 12: VeriPipe recovery example with region stitching

When reaching the end of *Rg*234, *Rg*1 moves to the *Commit* and is verified to be error-free, *Rg*234 enters into the *Verify*, and *Rg*5 starts with the *Execute*. Here, *Recovery PC* is updated with *Rg*1's end, while *Region Register* points to *Rg*234's end, as shown in Figure 12 (b). That way, upon a soft error detected in *Rg*5, VeriPipe guarantees the program to be recoverable from the beginning of *Rg*234. The reason is three-fold: (1) VeriPipe only uses the live-in registers of *Rg*234 to recover the program; (2) all of them are live-out from the prior verified regions and must already be error-free; (3) region stitching only makes certain checkpoints in *Rg*2 − *Rg*4 redundant, which do not affect *Rg*234's live-in registers.

Eventually, *Rg*234 is verified and transits to the *Commit* at the end of *Rg*5. *Rg*5 then moves to the *Verify*, and *Rg*6 starts with the *Execute*; VeriPipe updates its three registers and the countdown timer accordingly. Upon a soft error detected in *Rg*6, as shown in Figure 12 (c), VeriPipe runtime can successfully resume the program execution from *Rg*5's beginning—it is also the end of the verified region *Rg*234. The reason is twofold: (1) *Rg*5's live-in registers are defined in prior regions and thus already be verified to be error-free; (2) region stitching still reserves the checkpoints in *Rg*2-*Rg*4 for recovering *Rg*5.

7 DISCUSSION

7.1 Fault Model

VeriPipe assumes that the memory system (*i.e.,* caches and DRAM) is already protected with error-correcting code (ECC) as in commodity processors [11]. Also, VeriPipe assumes that its proposed hardware structures, *e.g.,* three registers and one timer, and store queue are hardened against soft errors as in prior designs [61, 69].

7.2 Why No Fault Injection?

As stated in [50, 63], soft errors are predominantly generated by energetic particle strikes that always generate a sound wave. Due to the physical phenomena, the sound wave must be detected by deployed acoustic sensors. Consequently, soft errors are sure to be detected [61] within a bounded latency no matter how many soft errors occur simultaneously, leading to never missed soft errors. Thus, the 100% guaranteed detection of particle-induced soft errors obviates the need for fault injections. Thanks to the near-zero overhead, VeriPipe can be integrated with other techniques [54] to achieve a full protection against all kinds of soft errors.

7.3 False Positive Rate

As prior work [61] shows, with calibration, acoustic sensors can avoid the detection of those particle strikes which do not generate bit flips, thus reducing the chance of reporting such weak strikes to zero. Nevertheless, false-positive case still occurs since not every bit flip causes a program failure, *e.g.,* incorrect program output, program crash, and program hang, because of the so-called bit-masking effect. If acoustic sensors do not trigger the detection of weak particle strikes, the false positive rate becomes the same as bit masking rate. According to prior studies [22, 23], the bit masking rate of soft errors is ≈90% for CPU applications, and Gupta *et al.* [22] found the post-masking failure rate is typically 0.9 error per day. Given all this, the pre-masking error rate per day is $\frac{0.9}{1-0.9} = 9.0$. Therefore, acoustic sensors are expected to report $9.0 \times 0.9 = 8.1$ errors per day. The implication is that VeriPipe runtime re-executes a region to correct a soft error occurred therein every ≈3 hours, in which case the overhead is negligible considering that the average region execution time is 47.63 *ns* (see Section 8.5).

7.4 Error Recovery for Multi-Cores

To ensure program recovery for multi-cores, VeriPipe assumes data-race-freedom (DRF) program which is guaranteed by C/C++ 11 standard [49] onwards. As with prior proposals [43, 69], VeriPipe treats synchronization primitives, *e.g.,* atomic operations and memory fences, as region boundaries such that critical sections form regions. The implication is three-fold: (1) the stores of critical sections are released to the memory and thus visible to other cores only after their verification; (2) upon detecting soft errors, there must be at most one core in a critical section since other cores have not obtained the lock of the section; (3) in the case of soft error detected, the cores can be independently rollbacked to their latest verified points without the need of tracking inter-thread dependence.

7.5 Exception and Interrupt

Upon detecting a soft error while receiving an exception/interrupt signal, VeriPipe resumes the program execution from the end of the latest verified region and continues the execution till the program point where the exception took place, After that, VeriPipe invokes the corresponding exception handler to deal with the specified exception as a regular processor does. Notably, VeriPipe has a minimal impact on the performance of exception handling since the chance of encountering both soft error and exception at the same time is practically negligible. Even if this case occurs, VeriPipe delays the exception handling by only 47.63 *ns* on average.

8 EVALUATION AND ANALYSIS

We implement our compiler optimizations atop Clang/LLVM 13 compiler [37] with about 2300 lines of code. All evaluated C/C++ applications are compiled with -O3 flag and statically linked. We implement our hardware design using gem5 [8] simulator to model an eight-core *out-of-order* ARMv8 Cortex-A77 processor [38] clocked at 2.42GHz. Each core has 256 reorder buffer (ROB) entries, 85 load queue (LQ) entries, 90 store queue (SQ) entries, 160 instruction queue (IQ) entries, and 256 physical registers. Also, each core is equipped with a 64KB 4-way private L1 data cache with 4-cycle hit latency and a 512KB 8-way private L2 cache with 9-cycle hit latency. All the eight cores share a 4MB 16-way L3 cache with 31-cycle hit latency. The main memory is configured to 16GB DDR4 2400 8x8. We treat the original program running on the original hardware platform without soft error resilience as our baseline.

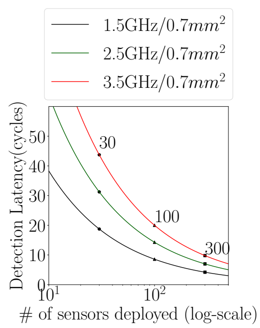

Figure 13: Detection latency across the number of sensors deployed

We run multi-threaded benchmarks, *e.g.,* SPLASH3 [56] and NPB-CPP [46], on gem5 full system (FS) mode, while simulating SPEC2006/2017 [9, 24]/Mini-Apps [29, 60] on system call emulation (SE) mode. We synchronize the number of simulated instructions by measuring the number of function calls in the baseline which is a constant across different binary version generated by various compiler optimizations. As with prior techniques [43, 68, 69], all SPEC/Mini-Apps applications are chosen to be fast forwarded 5 billion instructions with an atomic CPU, and then are simulated for 1 billion instructions in O3 CPU model. The FS simulation boots an Ubuntu 14.04 with Linux kernel 4.18.0 and runs SPLASH3/NPB-CPP with eight cores by default.

As WCDL is affected by the number of sensors deployed, we calculate the number of desired sensors for the given WCDL cycles as with prior work [43, 69]. Figure 13 shows that deploying 30-300 sensors achieves 50-10 cycles of WCDL on an ARM Cortex-A77 core—0.7 mm^2 core size excluding caches with TSMC 7*nm* technology—with varying clock frequency. With this in mind, we conservatively set the default WCDL to 30 cycles.

8.1 Run-time Overhead Analysis

To demonstrate the high performance of VeriPipe, we compare VeriPipe to the state-of-the-art work **Turnstile** [43] across a variety of WCDLs. We also apply VeriPipe compiler optimizations to Turnstile, forming a scheme called **Turnstile+VeriPipe Compiler**. We further implement the fast store release of Turnpike [69]—which proposes a hardware-based fast store release to bypass store verification and thus relieves the pressure on SQ, represented as **Turnstile+VeriPipe Compiler+Fast Store Release**.

As shown in Figure 14, VeriPipe is greatly superior to all prior techniques across all WCDLs from 10 to 50 cycles. VeriPipe incurs an average of only 1% run-time overhead for all the WCDLs, while Turnstile incurs an average of 5% to 14% and up to a 62% run-time overhead for the varying WCDLs. Here, **Turnstile+VeriPipe Compiler** still results in an unacceptable overhead for certain applications, *e.g.,* 61% for `povray`, 28% for `fft`, and 33% for `lu-contig`, though our compiler optimizations can improve the performance of Turnstile owing to eliminating redundant checkpoints. The reason is twofold: (1) Turnstile cannot tolerate small regions due to limited region boundary buffer (RBB)—20 entries in our configuration; and (2) VeriPipe compiler fails to enlarge these small regions (see Section 5 for the discussion in details). As a result, Turnstile cannot scale up to longer WCDLs no matter if enabling VeriPipe compiler optimizations. Noteworthily, **Turnstile+VeriPipe Compiler+Fast Store Release** still does not help in improving the performance of **Turnstile+VeriPipe Compiler** at all, despite bypassing the verification for certain stores. This is because (1) out-of-order cores are equipped with 10x larger store queue (SQ) than in-order cores, and thus (2) holding stores in the SQ for verification has no extra pressure on the SQ as confirmed in Section 8.6.2.

8.2 Impact of VeriPipe's Optimizations

To investigate the effect of each VeriPipe optimization, we conduct a series of experiments with the optimizations enabled incrementally and present the results in Figure 15.

Enforcement: shows the run-time overhead of enabling the naive dynamic enforcement. The figure shows that this naive strategy causes a significant run-time overhead, *e.g.,* 8% on average and up to 50%, due to pipeline stalls incurred at the end of each short region.

Stitching: stands for the run-time overhead of enabling region stitching. As the figure shows, region stitching significantly reduces the run-time overhead compared to the naive enforcement. On average, region stitching incurs 3% run-time overhead. In particular, region stitching lowers the overhead of many applications, *e.g.,* `h264ref`, `povray`, `leela`, `fft`, and `ocean-contig`, to near-zero.

+LICM: shows the run-time overhead after incrementally enabling the LICM. It further lowers the run-time overheads of certain applications, *e.g.,* 1% reduction for `xalancbmk`, 2% reduction for `mcf` and `nab`, and 5% reduction for `radix`.

+LIVM: depicts that the LIVM lowers the average run-time overhead to only 2%. In particular, the LIVM lowers the overheads of certain applications to near zero, *e.g.,* `mcf`, `lu-config`, and `radiosity`, thanks to its ability to move checkpoints out of loops.

+Pruning: indicates the run-time overhead of enabling the optimal pruning further. The figure shows that the pruning reduces the average overhead to only 1% as it eliminates redundant checkpoint stores. Note that the pruning lowers the overheads of many applications to near zero, *e.g.,* `cg`, `mg`, `lu-config`, and `ocean-contig`.

+Unrolling (VeriPipe): stands for the overall run-time overhead VeriPipe incurs with all optimizations enabled. Eventually, VeriPipe

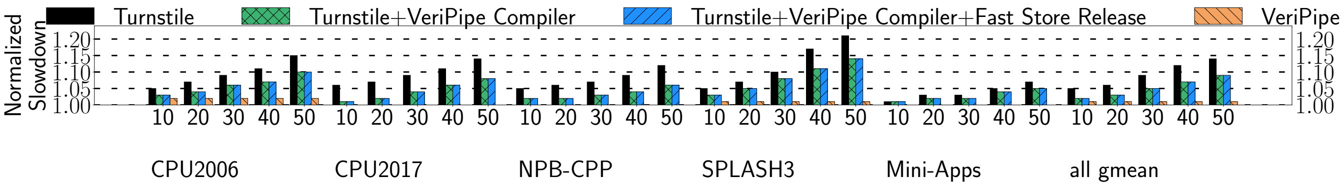

Figure 14: Run-time overhead comparison between VeriPipe and prior approaches with varying WCDL (default to 30 cycle); lower is better

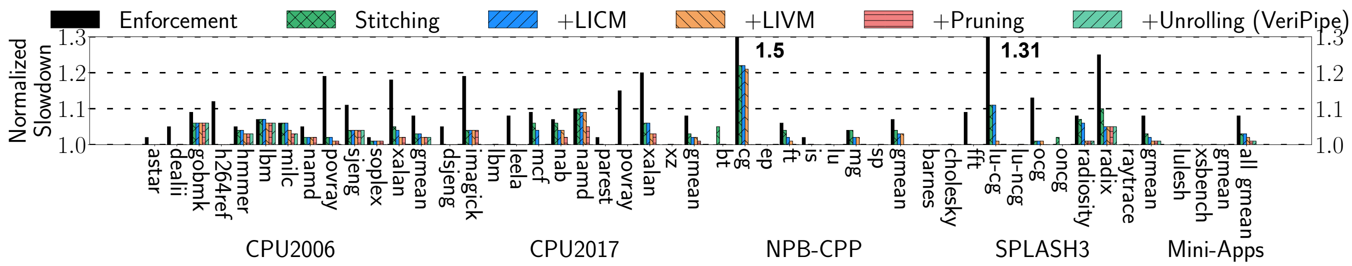

Figure 15: Impact of each VeriPipe optimization; lower is better

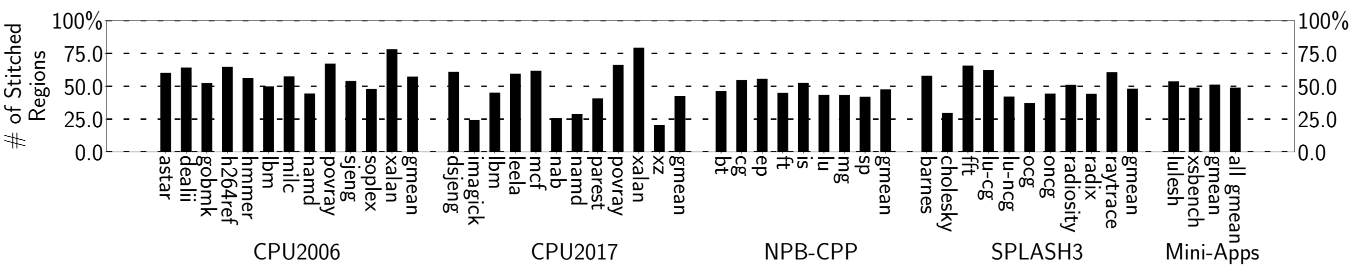

Figure 16: The ratio of the region eliminated by the region stitching to total regions; higher is better

incurs only 1% run-time overhead on average for total 43 applications. The figure shows that the unrolling can further reduce the overheads of some applications, *e.g.,* 2% reduction for namd, as it can avoid certain checkpoints of enlarged regions.

8.3 Effect of Region Stitching

To investigate the effectiveness of region stitching in eliminating region boundaries, we compute the ratio of the regions removed by the region stitching to total amount of regions. Figure 16 shows that the region stitching eliminates 49% of regions. This explains why the region stitching is so good at obviating pipeline stalls occurred at the end of short regions and lowering the run-time overhead.

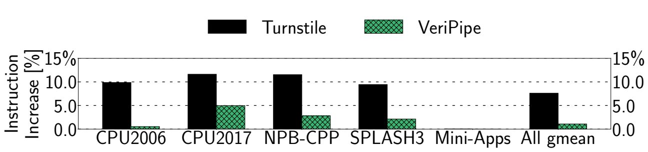

Figure 17: Normalized dynamic instruction increases of Turnstile and VeriPipe to the baseline; lower is better

8.4 Dynamic Instruction Increase

To inspect how effective VeriPipe's compiler optimizations are in eliminating redundant checkpoints, we collect the number of committed instructions of Turnstile and VeriPipe. Figure 17 shows that VeriPipe leads to an average of only 1.09% increase in committed instructions, while Turnstile incurs an average of 7.61% increase. Consequently, VeriPipe incurs minimal pressure on the instruction cache of modern server-class processors where the pressure has been becoming a concern [55].

8.5 Region Characteristics

To investigate why the state-of-the-art work Turnstile causes significant pipeline stalls at the end of regions, while VeriPipe incurs near-zero pipeline stalls at region end. We demonstrate the average region execution time of Turnstile and VeriPipe in Figure 18. We compute the region execution time by subtracting the commit time of the region's first instruction from that of the last instruction. Then, we average the execution time of all regions for each application. The figure shows that Turnstile's average region execution time is only 24.63 cycles—implying that Turnstile wastes many CPU cycles at the end of short regions, while VeriPipe's is 115.26 cycles thanks to the synergistic compiler/architecture codesign. Notably, owing to the region stitching, we can enlarge VeriPipe's region size further for longer WCDLs with no overhead incurred.

8.6 Sensitivity Analysis

8.6.1 Sensitivity to WCDL. To clearly show how WCDL affects VeriPipe's run-time performance, we test VeriPipe for varying WCDLs from 10 to 50 cycles as shown in Figure 19. The trend

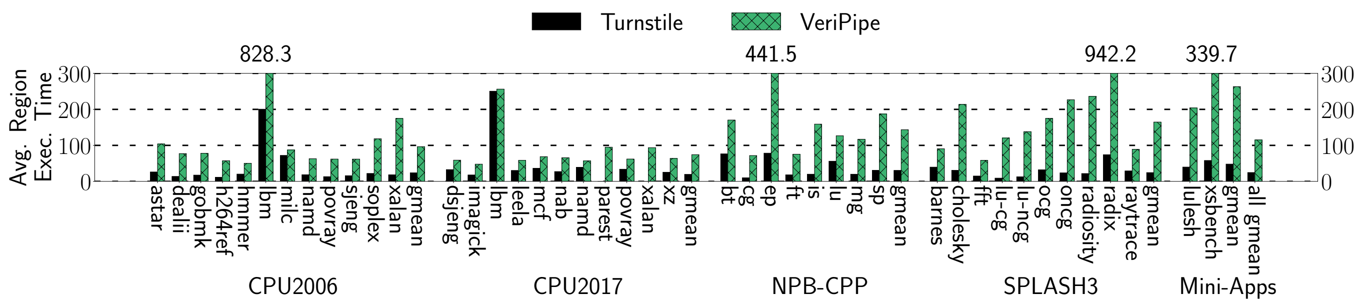

Figure 18: Average region execution time in cycles; higher is better for less CPU pipeline stalls at a region boundary

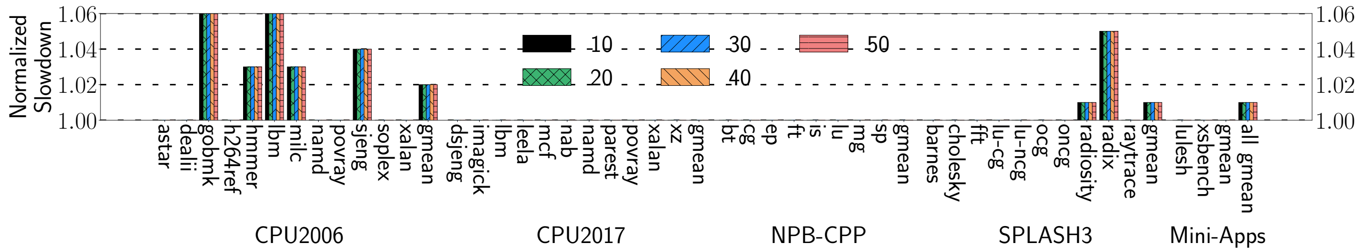

Figure 19: Normalized slowdown of VeriPipe with varying WCDL (default to 30); lower is better

is that VeriPipe incurs the same run-time overhead for all evaluated applications no matter how long the WCDL is owing to the simple yet effective region stitching. This implies that VeriPipe can significantly reduce the number of deployed acoustic sensors—lowering hardware complexity further—with no performance impact.

8.6.2 Sensitivity to Store Queue Size. You might think that buffering the data being stored in the store queue (SQ) for verification imposes extra pressure on the SQ. To see how this affects the run-time performance of VeriPipe, we vary the SQ size from 56 up to 110, which represent the SQ sizes of four typical high-performance out-of-order cores, *e.g.,* Marvell ThunderX3 [59], ARM Cortex-A76 [39], ARM Cortex-A77 [38], and Apple M1 [31]. Figure 20 shows that VeriPipe leads to no increase in the run-time overhead. This is because VeriPipe puts minimal pressure on the SQ owing to its compiler optimizations, allowing VeriPipe to be integrated into varying computing devices ranging from mobile platforms to datacenters.

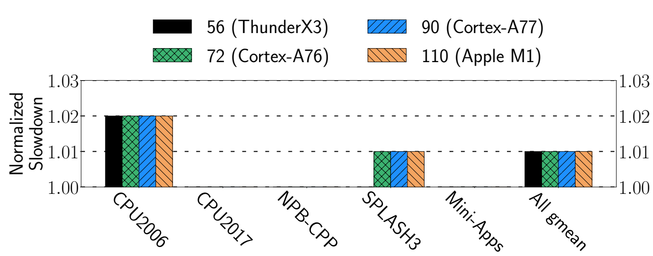

Figure 20: Normalized slowdown of VeriPipe to the baseline with varying SQ size from 56 up to 110; lower is better

8.6.3 Sensitivity to Thread Count. As with prior techniques [43, 69, 70], VeriPipe treats synchronization primitives as region boundaries for correct multi-cores recovery. This might delay the inter-thread synchronization due to adding verification latency to the execution delay of the synchronization primitives. To investigate such an impact, we vary the number of threads for NPB and SPLASH3 from 8 up to 64. As shown in Figure 21, VeriPipe practically has negligible (1%) impact on the performance of these program for all configurations. The reason is twofold: (1) critical sections account for a small portion of the program execution time; (2) VeriPipe incurs minimal stall cycles at the end of critical sections (regions) thanks to long enough regions.

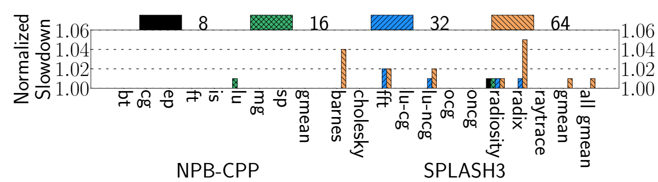

Figure 21: Normalized slowdown of VeriPipe with varying thread count from 8 to 64; lower is better

8.7 Power and Area Overheads

VeriPipe proposes only two 64-bit registers (*Recovery PC* and *Region Register*), a 7-bit register (*GSQ Pointer*), a 5-bit countdown timer, and a simple control logic comprised of a few comparators. We implement the hardware components of Turnstile/VeriPipe using Chisel [6] and compile the code into RTL with TSMC 28*nm*—the open-source RTL compiler we get only supports 28*nm*. Table 1 shows that VeriPipe incurs 8.8% area, 8.7% power consumption, and 26.9% access latency of Turnstile.

Table 1: Hardware cost comparison between Turnstile and VeriPipe with TSMC 28*nm* technology

	Area (μm^2)	Power (mW)	Max. Access latency (ns)
VeriPipe	849.1	2.4	0.07
Turnstile	9667.3	27.5	0.26
VeriPipe / Turnstile	8.8%	8.7%	26.9%

Due to the lack of open-source RTL compiler that supports TSMC 7*nm* node, we scale the synthesis results of the 28*nm* down by 6.6x [64, 65] to approximate the results of TSMC 7*nm* technology. It turns out that VeriPipe occupies only 0.018% area of a 0.7mm^2 ARM Cortex-A77 core (excluding caches).

9 OTHER RELATED WORK

Many prior techniques [18–20, 36, 54, 57, 58] propose to leverage instruction-level/thread-level/process-level duplication to detect the occurrences of soft errors, ending up with high run-time overhead. Although hardware-based techniques [4, 5, 35] can lower run-time overhead, they come with expensive hardware costs. Other schemes [62, 66] detect the error occurrence by checking for the symptoms that soft errors generate, while dual/triple modular redundancy (DMR/TMR) schemes [51, 53] check for faulty consequences. However, they all suffer from lower detection coverage. Prior recovery schemes [12–17, 25, 27, 34, 41, 67, 70, 72] cause high run-time overhead regardless of their recovery granularity. Recently, Kim *et al.* proposed Penny [32] to provide high-performance soft error resilience for GPUs. Penny makes use of parity code for immediate soft error detection and idempotent processing [17, 40, 42, 44] for error recovery. Flame [71] leverages acoustic sensors and idempotent processing for GPU soft error resilience. It exploits the massive multi-threading of GPUs to hide the verification latency of warps, achieving lightweight resilience.

10 CONCLUSION

This paper presents VeriPipe, a near-zero-overhead resilience scheme that protects out-of-order cores against soft errors with acoustic-sensor-based detection. VeriPipe compiler partitions input program into a series of recoverable regions, while VeriPipe hardware verifies whether they are error-free at run time. For the verification purpose, VeriPipe delays the writeback of any data stored during region execution until the region is verified to be error-free. If an error is detected, VeriPipe corrects it by resuming the program from the end of the latest verified (error-free) region. The experiments with 43 applications of SPEC 2006/2017/ NPB-CPP/SPLASH3/Mini-Apps demonstrate that VeriPipe incurs only a 1% run-time overhead, on average. In particular, VeriPipe achieves such high-performance soft error resilience at minimal hardware complexity (3 registers and 1 countdown timer), unlike the state-of-the-art work that requires intrusive microarchitectural modifications and yet causes a significant run-time overhead. We believe that VeriPipe lays the foundation for the commercialization of acoustic-sensor-based soft error protection.

ACKNOWLEDGMENTS

We thank anonymous reviewers for their valuable comments. This work was supported by NSF grants 2001124, 2153749, and 2314681.

REFERENCES

[1] Vikas Agarwal, MS Hrishikesh, Stephen W Keckler, and Doug Burger. 2000. Clock rate versus IPC: The end of the road for conventional microarchitectures. In *Proceedings of the 27th annual international symposium on Computer architecture.* 248–259.

[2] Dimitris Agiakatsikas, George Papadimitriou, Vasileios Karakostas, Dimitris Gizopoulos, Mihalis Psarakis, Camille Belanger-Champagne, and Ewart Blackmore. 2023. Impact of Voltage Scaling on Soft Errors Susceptibility of Multicore Server CPUs. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture.* 957–971.

[3] Alfred V Aho, Ravi Sethi, and Jeffrey D Ullman. 1986. Compilers, principles, techniques. *Addison wesley* 7, 8 (1986), 9.

[4] Sam Ainsworth and Timothy M Jones. 2018. Parallel error detection using heterogeneous cores. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN).* IEEE, 338–349.

[5] Sam Ainsworth and Timothy M Jones. 2019. Paramedic: Heterogeneous parallel error correction. In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN).* IEEE, 201–213.

[6] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. 2012. Chisel: constructing hardware in a scala embedded language. In *DAC Design Automation Conference 2012.* IEEE, 1212–1221.

[7] David F Bacon. 2022. Detection and Prevention of Silent Data Corruption in an Exabyte-scale Database System. (2022).

[8] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, et al. 2011. The gem5 simulator. *ACM SIGARCH Computer Architecture News* 39, 2 (2011), 1–7.

[9] James Bucek, Klaus-Dieter Lange, et al. 2018. Spec cpu2017: Next-generation compute benchmark. In *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering.* ACM, 41–42.

[10] Jonathan Chang, Yen-Huei Chen, Wei-Min Chan, Sahil Preet Singh, Hank Cheng, Hidehiro Fujiwara, Jih-Yu Lin, Kao-Cheng Lin, John Hung, Robin Lee, et al. 2017. 12.1 a 7nm 256mb sram in high-k metal-gate finfet technology with write-assist circuitry for low-v min applications. In *2017 IEEE International Solid-State Circuits Conference (ISSCC).* IEEE, 206–207.

[11] Wei Chen, Szu-Liang Chen, Siufu Chiu, Raghuraman Ganesan, Venkata Lukka, Wei Wing Mar, and Stefan Rusu. 2013. A 22nm 2.5 MB slice on-die L3 cache for the next generation Xeon® processor. In *2013 Symposium on VLSI Circuits.* IEEE, C132–C133.

[12] Jongouk Choi, Hyunwoo Joe, and Changhee Jung. 2022. CapOS: Capacitor Error Resilience for Energy Harvesting Systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 41, 11 (2022), 4539–4550.

[13] Jongouk Choi, Hyunwoo Joe, Yongjoo Kim, and Changhee Jung. 2019. Achieving stagnation-free intermittent computation with boundary-free adaptive execution. In *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS).* IEEE, 331–344.

[14] Jongouk Choi, Larry Kittinger, Qingrui Liu, and Changhee Jung. 2022. Compiler-directed high-performance intermittent computation with power failure immunity. In *2022 IEEE 28th Real-Time and Embedded Technology and Applications Symposium (RTAS).* IEEE, 40–54.

[15] Jongouk Choi, Qingrui Liu, and Changhee Jung. 2019. CoSpec: Compiler directed speculative intermittent computation. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture.* 399–412.

[16] Jongouk Choi, Jianping Zeng, Dongyoon Lee, Changwoo Min, and Changhee Jung. 2023. Write-light cache for energy harvesting systems. In *Proceedings of the 50th Annual International Symposium on Computer Architecture.* 1–13.

[17] Marc De Kruijf and Karthikeyan Sankaralingam. 2013. Idempotent code generation: Implementation, analysis, and evaluation. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO).* IEEE Computer Society, 1–12.

[18] Moslem Didehban and Aviral Shrivastava. 2016. nZDC: A compiler technique for near zero silent data corruption. In *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC).* IEEE, 1–6.

[19] 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.

[20] Moslem Didehban, Aviral Shrivastava, and Sai Ram Dheeraj Lokam. 2017. NEMESIS: A software approach for computing in presence of soft errors. In *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD).* IEEE, 297–304.

[21] James Elliott, Mark Hoemmen, and Frank Mueller. 2014. Tolerating silent data corruption in opaque preconditioners. *arXiv preprint arXiv:1404.5552* (2014).

[22] Saurabh Gupta, Tirthak Patel, Christian Engelmann, and Devesh Tiwari. 2017. Failures in large scale systems: long-term measurement, analysis, and implications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis.* 1–12.

[23] Siva Kumar Sastry Hari, Sarita V Adve, and Helia Naeimi. 2012. Low-cost program-level detectors for reducing silent data corruptions. In *IEEE/IFIP international conference on dependable systems and networks (DSN 2012).* IEEE, 1–12.

[24] John L Henning. 2006. SPEC CPU2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News* 34, 4 (2006), 1–17.

[25] Shao-Yu Huang, Jianping Zeng, Xuanliang Deng, Sen Wang, Ashrarul Haq Sifat, Burhanuddin Bharmal, Jiabin Huang, Ryan Williams, Haibo Zeng, and Changhee Jung. 2023. RTailor: Parameterizing Soft Error Resilience for Mixed-Criticality Real-Time Systems. In *2023 IEEE Real-Time Systems Symposium (RTSS).* IEEE.

[26] Jungi Jeong, Jaewan Hong, Seungryoul Maeng, Changhee Jung, and Youngjin Kwon. 2020. Unbounded hardware transactional memory for a hybrid DRAM/NVM memory system. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 525–538.

[27] Jungi Jeong and Changhee Jung. 2021. PMEM-spec: persistent memory speculation (strict persistency can trump relaxed persistency). In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 517–529.

[28] Jungi Jeong, Jianping Zeng, and Changhee Jung. 2022. Capri: Compiler and architecture support for whole-system persistence. In *Proceedings of the 31st International Symposium on High-Performance Parallel and Distributed Computing*. 71–83.

[29] Ian Karlin, Jeff Keasler, and J Robert Neely. 2013. *Lulesh 2.0 updates and changes*. Technical Report. Lawrence Livermore National Lab.(LLNL), Livermore, CA.

[30] Himanshu Kaul, Mark Anders, Steven Hsu, Amit Agarwal, Ram Krishnamurthy, and Shekhar Borkar. 2012. Near-threshold voltage (NTV) design: Opportunities and challenges. In *Proceedings of the 49th Annual Design Automation Conference*. 1153–1158.

[31] Connor Kenyon and Collin Capano. 2022. Apple Silicon Performance in Scientific Computing. In *IEEE High Performance Extreme Computing Conference*. 1–10.

[32] Hongjune Kim, Jianping Zeng, Qingrui Liu, Mohammad Abdel-Majeed, Jaejin Lee, and Changhee Jung. 2020. Compiler-directed soft error resilience for lightweight GPU register file protection. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 989–1004.

[33] Kevin Krewell. 2012. Cortex-A53 Is ARM's next little thing. *Microprocessor Report* 11, 5 (2012), 12–2.

[34] Dmitrii Kuvaiskii, Rasha Faqeh, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. 2016. HAFT: Hardware-assisted fault tolerance. In *Proceedings of the Eleventh European Conference on Computer Systems*. 1–17.

[35] Christopher LaFrieda, Engin Ipek, Jose F Martinez, and Rajit Manohar. 2007. Utilizing dynamically coupled cores to form a resilient chip multiprocessor. In *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'07)*. IEEE, 317–326.

[36] Ignacio Laguna, Martin Schulz, David F Richards, Jon Calhoun, and Luke Olson. 2016. Ipas: Intelligent protection against silent output corruption in scientific applications. In *2016 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 227–238.

[37] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004*. IEEE, 75–86.

[38] ARM Limited. 2019. ARM Cortex A77. "https://www.arm.com/products/silicon-ip-cpu/cortex-a/cortex-a77".

[39] ARM limited Corporation. 2019. Cortex-a76 technique reference manual. https://developer.arm.com/Processors/Cortex-A76.

[40] Qingrui Liu, Joseph Izraelevitz, Se Kwon Lee, Michael L Scott, Sam H Noh, and Changhee Jung. 2018. iDO: Compiler-directed failure atomicity for nonvolatile memory. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 258–270.

[41] Qingrui Liu and Changhee Jung. 2016. Lightweight hardware support for transparent consistency-aware checkpointing in intermittent energy-harvesting systems. In *2016 5th Non-Volatile Memory Systems and Applications Symposium (NVMSA)*. IEEE, 1–6.

[42] Qingrui Liu, Changhee Jung, Dongyoon Lee, and Devesh Tiwari. 2016. Compiler-directed lightweight checkpointing for fine-grained guaranteed soft error recovery. In *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 228–239.

[43] Qingrui Liu, Changhee Jung, Dongyoon Lee, and Devesh Tiwari. 2016. Low-cost soft error resilience with unified data verification and fine-grained recovery for acoustic sensor based detection. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Press, 25.

[44] Qingrui Liu, Changhee Jung, Dongyoon Lee, and Devesh Tiwari. 2017. Compiler-directed soft error detection and recovery to avoid DUE and SDC via Tail-DMR. *ACM Transactions on Embedded Computing Systems (TECS)* 16, 2 (2017), 32.

[45] Jack L Lo and Susan J Eggers. 1995. Improving balanced scheduling with compiler optimizations that increase instruction-level parallelism. *ACM SIGPLAN Notices* 30, 6 (1995), 151–162.

[46] Júnior Löff, Dalvan Griebler, Gabriele Mencagli, Gabriell Araujo, Massimo Torquati, Marco Danelutto, and Luiz Gustavo Fernandes. 2021. The NAS parallel benchmarks for evaluating C++ parallel programming frameworks on shared-memory architectures. *Future Generation Computer Systems* 125 (2021), 743–757.

[47] Doug Matzke. 1997. Will physical scalability sabotage performance gains? *Computer* 30, 9 (1997), 37–39.

[48] Pablo Montesinos, Wei Liu, and Josep Torrellas. 2007. Using register lifetime predictions to protect register files against soft errors. In *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'07)*. IEEE, 286–296.

[49] Robin Morisset, Pankaj Pawan, and Francesco Zappa Nardelli. 2013. Compiler testing via a theory of sound optimisations in the C11/C++ 11 memory model. *ACM SIGPLAN Notices* 48, 6 (2013), 187–196.

[50] Shubhendu S Mukherjee, Joel Emer, and Steven K Reinhardt. 2005. The soft error problem: An architectural perspective. In *11th International Symposium on High-Performance Computer Architecture*. IEEE, 243–247.

[51] Shubhendu S Mukherjee, Michael Kontz, and Steven K Reinhardt. 2002. Detailed design and evaluation of redundant multi-threading alternatives. In *Proceedings 29th annual international symposium on computer architecture*. IEEE, 99–110.

[52] George Papadimitriou and Dimitris Gizopoulos. 2023. Silent Data Corruptions: Microarchitectural Perspectives. *IEEE Trans. Comput.* (2023), 1–13. https://doi.org/10.1109/TC.2023.3285094

[53] Joydeep Ray, James C Hoe, and Babak Falsafi. 2001. Dual use of superscalar datapath for transient-fault detection and recovery. In *Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture*. IEEE Computer Society, 214–224.

[54] George A Reis, Jonathan Chang, Neil Vachharajani, Ram Rangan, and David I August. 2005. SWIFT: Software implemented fault tolerance. In *Proceedings of the international symposium on Code generation and optimization*. IEEE Computer Society, 243–254.

[55] Alberto Ros and Alexandra Jimborean. 2020. The entangling instruction prefetcher. *IEEE Computer Architecture Letters* 19, 2 (2020), 84–87.

[56] Christos Sakalis, Carl Leonardsson, Stefanos Kaxiras, and Alberto Ros. 2016. Splash-3: A properly synchronized benchmark suite for contemporary research. In *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 101–111.

[57] Hwisoo So, Moslem Didehban, Yohan Ko, Aviral Shrivastava, and Kyoungwoo Lee. 2018. EXPERT: Effective and flexible error protection by redundant multithreading. In *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 533–538.

[58] Hwisoo So, Moslem Didehban, Aviral Shrivastava, and Kyoungwoo Lee. 2019. A software-level redundant multithreading for soft/hard error detection and recovery. In *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 1559–1562.

[59] Rabin Sugumar, Mehul Shah, and Ricardo Ramirez. 2021. Marvell ThunderX3: Next-Generation Arm-Based Server Processor. *IEEE Micro* 41, 2 (2021), 15–21.

[60] John R Tramm, Andrew R Siegel, Tanzima Islam, and Martin Schulz. 2014. XSBench-the development and verification of a performance abstraction for Monte Carlo reactor analysis. *The Role of Reactor Physics toward a Sustainable Future (PHYSOR)* (2014).

[61] Gaurang R Upasani. 2016. *Soft error mitigation techniques for future chip multiprocessors*. Ph. D. Dissertation. Universitat Politècnica de Catalunya.

[62] Nicholas J Wang and Sanjay J Patel. 2006. ReStore: Symptom-based soft error detection in microprocessors. *Dependable and Secure Computing, IEEE Transactions on* 3, 3 (2006), 188–201.

[63] Christopher Weaver, Joel Emer, Shubhendu S Mukherjee, and Steven K Reinhardt. 2004. Techniques to reduce the soft error rate of a high-performance microprocessor. *ACM SIGARCH Computer Architecture News* 32, 2 (2004), 264.

[64] Shien-Yang Wu, CY Lin, MC Chiang, JJ Liaw, JY Cheng, SH Yang, CH Tsai, PN Chen, T Miyashita, CH Chang, et al. 2016. A 7nm CMOS platform technology featuring 4 th generation FinFET transistors with a 0.027 um 2 high density 6-T SRAM cell for mobile SoC applications. In *2016 IEEE International Electron Devices Meeting (IEDM)*. IEEE, 2–6.

[65] Shien-Yang Wu, Colin Yu Lin, MC Chiang, JJ Liaw, JY Cheng, SH Yang, Ming Liang, Tadakazu Miyashita, CH Tsai, BC Hsu, et al. 2013. A 16nm FinFET CMOS technology for mobile SoC and computing applications. In *2013 IEEE International Electron Devices Meeting*. IEEE, 9–1.

[66] Jing Yu, Maria Jesus Garzaran, and Marc Snir. 2009. Esoftcheck: Removal of non-vital checks for fault tolerance. In *2009 International Symposium on Code Generation and Optimization*. IEEE, 35–46.

[67] Jianping Zeng, Jongouk Choi, Xinwei Fu, Ajay Paddayuru Shreepathi, Dongyoon Lee, Changwoo Min, and Changhee Jung. 2021. ReplayCache: Enabling Volatile Cachesfor Energy Harvesting Systems. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. 170–182.

[68] Jianping Zeng, Jungi Jeong, and Changhee Jung. 2023. Persistent Processor Architecture. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*. 1075–1091.

[69] Jianping Zeng, Hongjune Kim, Jaejin Lee, and Changhee Jung. 2021. Turnpike: Lightweight Soft Error Resilience for In-Order Cores. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. 654–666.

[70] Jianping Zeng, Tong Zhang, and Changhee Jung. 2024. Compiler-Directed Whole-System Persistence. In *Proceedings of the 51th Annual International Symposium on Computer Architecture*.

[71] Yida Zhang and Changhee Jung. 2022. Featherweight soft error resilience for GPUs. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 245–262.

[72] Yuchen Zhou, Jianping Zeng, Jungi Jeong, Jongouk Choi, and Changhee Jung. 2023. SweepCache: Intermittence-Aware Cache on the Cheap. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*. 1059–1074.