Turnpike: Lightweight Soft Error Resilience for In-Order Cores

Jianping Zeng
Purdue University
USA

Hongjune Kim
Seoul National University
Korea

Jaejin Lee
Seoul National University
Korea

Changhee Jung
Purdue University
USA

ABSTRACT

Acoustic-sensor-based soft error resilience is particularly promising, since it can verify the absence of soft errors and eliminate silent data corruptions at a low hardware cost. However, the state-of-the-art work incurs a significant performance overhead for in-order cores due to frequent structural/data hazards during the verification. To address the problem, this paper presents Turnpike, a compiler/architecture co-design scheme that can achieve lightweight yet guaranteed soft error resilience for in-order cores. The key idea is that many of the data computed in the core can bypass the soft error verification without compromising the resilience. Along with simple microarchitectural support for realizing the idea, Turnpike leverages compiler optimizations to further reduce the performance overhead. Experimental results with 36 benchmarks demonstrate that Turnpike only incurs a 0-14% run-time overhead on average while the state-of-the-art incurs a 29-84% overhead when the worst-case latency of the sensor based error detection is 10-50 cycles.

ACM Reference Format:
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 (MICRO '21), October 18–22, 2021, Virtual Event, Greece.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3466752.3480042

1 INTRODUCTION

Soft error resilience is becoming more important than ever. With technology scaling, circuits are likely to be more sensitive to radiation-induced soft errors; they are mostly caused by energetic particles (e.g., cosmic rays) and alpha particles from packaging materials [15, 27, 47, 67–70]. Soft errors may lead to a system crash or even worse silent data corruptions (SDC) that are not caught by the error detection logic but end up with incorrect outputs. Due to the high availability requirement of embedded systems, soft error resilience has been one of the most important design considerations.

Among existing soft error resilience schemes, acoustic-sensor-based detection [8, 34, 37, 39, 40, 51, 67–72] is particularly promising. To the best of our knowledge, it is the only way to prevent SDC—that is a long-awaited open problem—at a low hardware cost. Since acoustic sensors perceive the sound wave of particle strikes, which is always generated as a physical phenomenon, no resulting soft error is missed. As such, the sensor-based detection can achieve SDC freedom; unlike other schemes, it does not even require any microarchitecture replication. Moreover, sensors occupy only a very small die size area. For example, 300 sensors are enough to achieve 30 cycles of the worst-case detection latency (WCDL) for a 2GHz out-of-order core, and they only cause ∼1% area overhead [67–72].

With that in mind, Liu *et al* [39] show how their solution Turnstile can leverage acoustic sensors for core-level error containment with little architecture change for soft error verification/recovery. The rationale for verifying the absence of soft errors is that since each error is to be detected within WCDL after its occurrence, execution prior to a given time T will be verified to be error-free at a time T+WCDL, if no error is detected during the WCDL.

In light of this, Turnstile verifies every data being stored to memory, ensuring that it has not been affected by soft errors before its write-back. Although a re-order buffer (ROB) retires a store instructions, the data is not written back to memory but held in a store buffer until it turns out to be verified waiting for WCDL. For register verification, Turnstile reformulates it based on the aforementioned memory verification by inserting stores to checkpoint updated live-out registers and holding them in the store buffer. The upshot is that register write-backs are never delayed for verification, which would otherwise slow down the pipeline execution. Since stores are rarely on the critical path in out-of-order cores, Turnstile can offer lightweight soft error resilience at ≈8% performance overhead on average for SPEC2006/MediaBench/SPLASH2 benchmarks.

Compared to out-of-order (OoO) cores, however, there has been less attention received to enhance the reliability of in-order cores in a low-cost manner—though they are widely deployed in embedded systems to control the physical world. For example, while in-order cores are used for adaptive cruise control, precrash safety alarm, and motion planning systems due to the simple hardware and the time predictability demand excluding complex OoO execution [19, 21, 43, 57, 58, 73], they still rely on expensive dual/triple modular redundancy (DMR/TMR) to deal with soft errors [3–5, 24–26, 42, 66]. Nonetheless, mission-critical embedded systems should pursue power-efficiency as they are often battery operated, e.g., portable military devices, wearables, and drones, preventing the use of DMR/TMR. Apart from that, DMR/TMR could suffer the size and weight issues that are particularly critical for tiny aerial systems such as spying drones [10, 49, 50] and bionic birds [12, 78].

With the increasing demand for lightweight soft error resilience for in-order cores, one might want to leverage Turnstile on top of in-order cores. Unfortunately, naively adapting Turnstile to in-order cores causes a significant performance overhead, i.e., 29%-84% for 10-50 cycles of WCDL. The main reason is that the in-order pipeline stalls for the structural/data hazards of stores due to the inability to schedule other independent instructions. Since the store buffer of in-order cores is very small (4 entries) unlike that of out-of-order cores (40 or more entries), it often becomes full during the verification, in which case the pipeline stalls on the next store due to the structural hazard until some of the buffered stores are

MICRO '21, October 18–22, 2021, Virtual Event, Greece

ACM ISBN 978-1-4503-8557-2/21/10.
https://doi.org/10.1145/3466752.3480042

verified and flushed to L1 cache. Similarly, for the execution of a checkpoint, i.e., essentially a store instruction to save a register value, the in-order pipeline may stall waiting for the value to be available. This data hazard happens a lot leading to significant performance degradation, because Turnstile inserts the checkpoint right after the register-update instruction, e.g., a delinquent load.

To address the problems, this paper presents Turnpike, a compiler/architecture co-design scheme that can achieve a lightweight yet guaranteed soft error resilience for in-order cores. Turnpike leverages 3 key insights to minimize the pipeline stalls with lowering the store buffer pressure. First, during code generation, it is possible to decrease the number of stores to be verified. Turnpike's compiler optimizations remove unnecessary checkpoint stores, e.g., those whose value can be reconstructed from other checkpointed values at the recovery time [29, 38], without compromising the recoverability. We also propose 2 novel compiler optimizations to suppress the generation of stores: loop induction variable merging for reducing live registers being checkpointed in a loop, and store-aware register allocation for less register-spilling stores.

Second, the compiler can reduce the execution delay of unremoved checkpoint stores with the help of instruction scheduling for resolving the checkpoint data hazard. That is, Turnpike attempts to separate the live register-update instructions from their dependent checkpoint stores by filling the gap with other independent instructions. This gives the in-order core an illusion that it can hide the execution delay of the checkpoint as in out-of-order execution.

Third, many of the remaining stores can be safely released to cache without waiting for verification, no matter if they are regular stores or checkpoint stores. For example, some value being stored is never used for the recovery of a soft error—even if it corrupts the value. To take advantage of this insight, we introduce simple hardware support that can (1) conduct the safety check for the fast (early) release of a given store and, if possible, (2) let it go through the fast path[1], i.e., immediately flushing it to cache bypassing its verification. Along with the above compiler optimizations, this hardware support can relieve the pressure on the small store buffer of in-order cores and thus reduce its structural hazards effectively. Experiments with 36 benchmarks from SPEC2006/2017/SPLASH3 suites highlight Turnpike's low performance overhead, i.e., 0% and 14% on average—while Turnstile's overhead is 29% and 84%—for 10 and 50 cycles of WCDL, respectively. Our contributions are below:

- Turnpike is the first to make acoustic-sensor-based soft error resilience work for in-order cores at a low HW/run-time cost.
- We show how compiler optimizations are used not only to remove unnecessary checkpoints but also to reduce the execution cycle of the unremoved checkpoints.
- We propose 2 novel compiler optimizations to lower the number of registers being checkpointed and reduce register-spilling stores during register allocation.
- We propose 2 new hardware schemes to bypass store verification without compromising resilience guarantee.

[1]The fast path can be regarded as an electronic toll collection lane on turnpike. The name Turnpike is inspired by this analogy.

2 BACKGROUND

This section describes how Turnstile, the state-of-the-art work, achieves lightweight sensor-based soft error verification with region-level error detection and recovery.

2.1 Region-Level Soft Error Verification

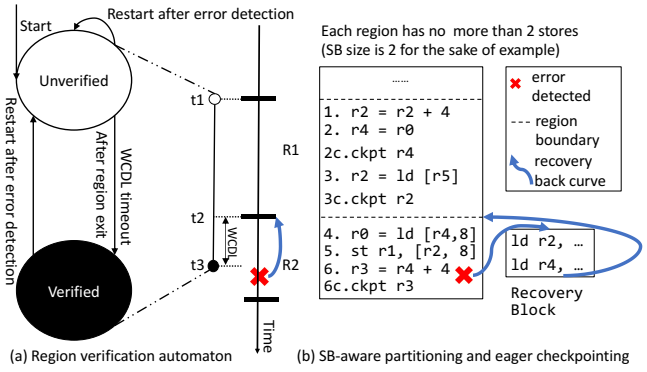

Figure 1: (a) Turnstile's region verification automaton; (b) store buffer aware region partitioning; eager checkpointing

To realize the sensor-based soft error verification at a low cost, Turnstile's compiler partitions the entire program into a series of verifiable/recoverable regions with the store buffer (SB) in mind so that each region cannot have more stores than the SB size [39]. As shown in Figure 1 (a), each started region is treated as unverified at the beginning, e.g., $R1$ gets $Unverified$ state at time $t1$. Thus, Turnstile prevents all the stores of the region from being merged to cache until the region is verified to be error-free. That is, no sensor detects an error during the worst-case detection latency (WCDL)—e.g., from $t2$ to $t3$—after the region is finished. That way Turnstile can contain all the errors occurred during the execution of a region within the core, keeping cache/memory intact. Furthermore, this allows Turnstile to correct an error by simply reading verified data from cache/memory protected by ECC in modern processors including even low-power cores such as ARM Cortex series.

For the in-core error containment,Turnstile leverages its SB as a gated store buffer (GSB) [9, 36]; hereafter, SB refers to GSB. That is, it holds by default all store write-backs for quarantine even after ROB retires the stores. To get them out of the SB quarantine, if verified (i.e., no error detected during WCDL time after the end of their region), Turnstile devises a region boundary buffer (RBB) shown in Figure 2. Whenever a region boundary is encountered, i.e., one region finishes and the next starts as at $t2$ in Figure 1 (a), Turnstile allocates the RBB entry to delineate the previously quarantined stores that will be released on their region verification. Especially when a region is verified, e.g., $R1$ at $t3$, the RBB marks the boundary, at which the verified region has ended, as a *recovery PC* in case of a future error.

2.2 Eager Checkpointing and Error Recovery

The Turnstile compiler performs so-called eager checkpointing [39] that immediately saves updated live-out registers in memory. That

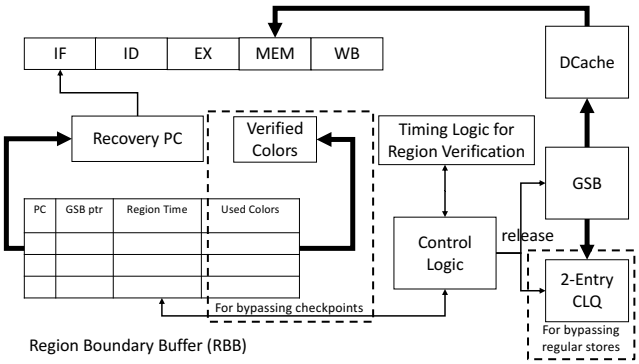

Figure 2: The high-level view of Turnpike; bold lines correspond to data paths while thin lines to control paths

is, it inserts a checkpoint store right after the register-update instruction provided the register is used as the input of later regions, e.g., at line 2c, 3c, and 6c in Figure 1 (b). The implication is three-fold. First, even if a region has multiple updates of a register, only the last one is checkpointed as the live-out register, e.g., Turnstile checkpoints only the definition of $r2$ at line 3 though it is pre-defined at line 1 in the figure. Second, Turnstile can turn register verification into memory verification; registers are verified through their checkpoint—which is essentially a store instruction—in the same way as stores are verified, without delaying any register write-back for performance reasons. Third, the checkpointed register values should be loaded to recover from a soft error.

Upon the detection of a soft error, Turnstile first discards all SB entries—because they could have been corrupted by the error—and identifies the most recently verified region boundary by referring to the *recovery PC* and the region starting thereafter. As shown in Figure 1 (b), Turnstile then executes the recovery block of the region to restore its input (live-in) registers, e.g., $r2$, $r4$ in the figure, from the ECC-protected memory (cache) where their checkpoints have been stored safely with the in-core error containment. Finally, Turnstile restarts the region recovering from the error. This soft error verification of Turnstile works well for out-of-order cores. However, it incurs a significant run-time overhead for in-order cores as shown in the next section.

3 MOTIVATION

This section discusses 3 main reasons why Turnstile, the state-of-the-art work, incurs a high run-time overhead for in-order cores: (1) Turnstile's checkpoints put significant pressure on the small store buffer (SB) leading to the structural hazards. (2) Once an SB entry is allocated, it stays long therein till the region is verified, which keeps holding the pressure and makes it take a while to resolve the structural hazards. (3) Due to eager checkpointing, the dependence between a register-update instruction and its immediate successor (i.e., a checkpoint store) often causes data hazards, slowing down the region execution. It is worth noting that the above problems are not a big deal for out-of-order cores thanks to the large SB (≧40) and the ability to schedule independent instructions to address the hazards. In contrast, the problems are devastating for in-order cores in that the SB has only a few entries (e.g., 4 in ARM Cortex-A53), and the hazards freeze all following instructions because of the in-order pipeline execution.

3.1 Checkpoint Puts Pressure on Store Buffer

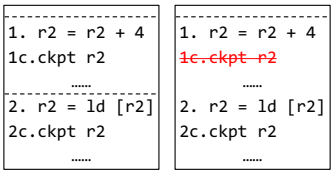

Figure 3: Impact of region size

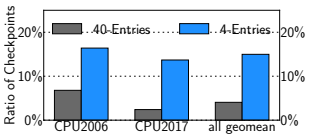

Figure 4: Impact of store buffer size on the number of inserted checkpoints

To equip the verifiable regions—partitioned with the SB size in mind—with the recoverability, Turnstile checkpoints live-out register values of each region by logging them to memory (Section 2). Since the regions are generally short due to the tiny SB (only 4 entries in modern in-order cores), the short regions tend to have more live-out registers updated overall than the long regions for a large SB. For example, while a register $r2$ is live-out in both regions and thus checkpointed twice at line 1c and 2c in Figure 3 (a), it is checkpointed only once in the same code that does not have a region boundary in-between as shown in Figure 3 (b); that is because $r2$ defined at line 1 is no longer live-out since it is overwritten by the following definition at line 2.

Figure 4 confirms that the number of inserted checkpoints (i.e., store instructions logging the live-out registers) significantly increases when the store buffer is shrunk from 40 to 4 entries. When the store buffer (SB) size is 40 as in out-of-order cores, Turnstile's eager checkpointing accounts for 4.1% of the total dynamic instruction count on average for SPEC 2006/2017 benchmark applications. On the other hand, when the SB size is 4 as in in-order cores, the ratio significantly increases to 14.98%. It turns out that such many checkpoints often fill up the SB, making the next store stall the pipeline due to the lack of room in the SB, i.e., the structural hazard. In particular, we found it possible to remove many of the checkpoints without compromising the soft error resilience; Section 4.1 discusses it in detail.

3.2 Verification Keeps the SB Pressure Long

To verify each region (ensuring the absence of soft errors during the region execution), Turnstile holds all the data being stored in the SB till the region is verified to be error-free. Hence, no allocated SB entries of stores can be released to L1 cache during the execution of their region; rather, they can only be released WCDL (10-50) cycles later after the region ends. The implication is that stores cannot but reside in the SB for such a long period of verification time, keeping the high pressure on the SB. In essence, this may cause a structural hazard if the SB has already been full when the pipeline encounters a new store, e.g., `inst N: st` in Figure 5. Unfortunately, the hazard cannot be resolved until the prior region is verified with its stores released to cache. In other words, the pipeline stall continues all the way to the region verification point—where the WCDL time elapses in the figure. Here, due to the in-order nature of the pipeline, it cannot schedule any of the following instructions thus being unable

to hide such a long stall latency. As such, it postpones not only the stalled store instruction, e.g., `inst N: st` in the figure, but also all the subsequent instructions. As will be shown in Section 4.3, Turnpike can safely release some stores from the SB without holding them for verification, thereby relieving the store buffer pressure.

3.3 Eager Checkpointing Slows Down the Store

Turnstile's eager checkpointing introduces a read-after-write dependence between the instruction, that updates a live-out register, and its checkpoint store. This is particularly harmful for the in-order pipeline because of the inability to dynamically schedule other independent instructions—unlike the out-of-order pipeline that can overlap their execution with the live-register-update instruction. In an in-order core, checkpoint stores can often be stalled since the register being checkpointed is not available for their execution.

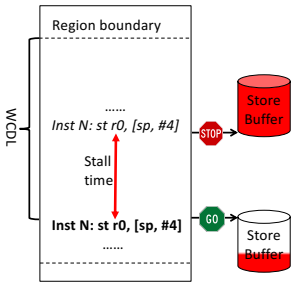

Figure 5: Stall due to the lack of room in the SB

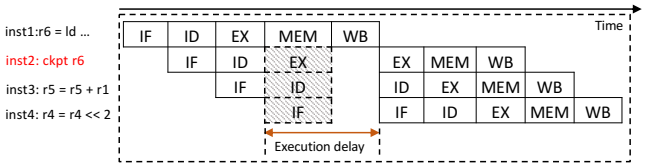

Figure 6: Checkpoint's execution delay on in-order pipeline

In such a case, the in-order pipeline must be delayed for a certain time to resolve the data hazard, e.g., till the value of register *r6* becomes available in Figure 6 where checkpoint store is marked in red. Since it is the load instruction that updates the *r6* in this example, the execution delay of the checkpoint store could be significant on cache misses. The takeaway is that such a checkpoint execution delay translates to the significant extension of the program execution time; Section 4.2 shows how Turnpike reduces the delay to make the checkpoint store instruction execute faster.

4 TURNPIKE FOR SOLVING THE PROBLEMS

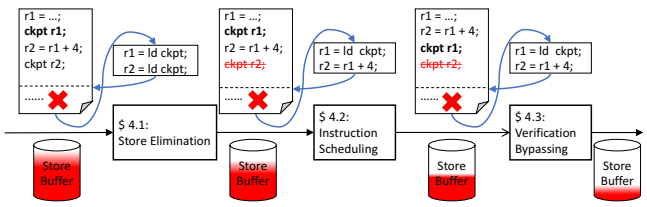

Figure 7: The 3 phases of Turnpike SW/HW optimizations

To address the 3 problems in Section 3, our proposal, Turnpike, leverages compiler and architectural optimizations; Figure 7 shows the workflow of the 3 optimization phases.

In the first phase (Section 4.1), Turnpike's 2 new compiler optimizations reduce the traffic to store buffer by (1) generating less spilling stores during register allocation and (2) eliminating a loop induction variable being checkpointed if it can be merged with others. Likewise, Turnpike removes unnecessary checkpoints with two existing compiler optimizations, checkpoint pruning [38] and loop invariant code motion (LICM) [46] to further lower the store buffer traffic. Second, for the remaining checkpoints that cannot be removed by the first phase, the Turnpike compiler performs checkpoint-aware instruction scheduling to hide the delay caused by data hazards (Section 4.2). Finally, to directly reduce the pressure on the store buffer (SB), Turnpike leverages 2 novel hardware techniques—in Section 4.3—that can (1) skip the verification of the remaining checkpoint stores and the regular stores and (2) merge them to cache right after they are committed. Note that the above 3 optimization phases have a synergistic impact on reducing the SB pressure. The rest of this section details the 3 optimizations.

4.1 Reducing the Traffic to the Store Buffer

This section shows how to reduce the SB traffic with 2 new compiler optimizations and 2 other existing ones. While the first addresses regular stores, the next 3 optimizations do checkpoint stores.

4.1.1 Store-Aware Register Allocation. In addition to application stores, the other source of regular stores is register allocation. Since it is done in a best effort manner, some variables end up being spilled to stack memory when architectural registers run out during the register allocation. To avoid spilling performance-critical variables, traditional register allocators takes a heuristic approach to determine what to spill. More precisely, they maintain the spill cost (weight) of variables which summarizes the execution frequency of their use points (reads and writes). Unfortunately, since the spill code models of traditional register allocators do not differentiate writes from reads, they may generate superfluous spilling stores. While this is not a concern for most processors where stores are off the critical path, in-order cores equipped with sensor based soft error verification can suffer a significant performance degradation. To address this problem, the Turnpike compiler increases the cost for the write operation of each variable in the spill candidate decision logic. Note that care must be taken to maintain the original register allocation quality in terms of the number of spilled variables, which would otherwise degrade the performance of the resulting code. As a result, Turnpike can keep those variables, that are frequently written, in architectural registers, and thus all the writes to the variables just become register writes other than memory stores.

4.1.2 Loop Induction Variable Merging (LIVM). We found that traditional compilers often generate additional loop induction variables—that must be checkpointed each loop iteration—and the resulting checkpoint stores increase the store buffer traffic significantly. There are two kinds of induction variables [46]: basic induction variable, e.g., *i* in Figure 8 (a) and induced induction variable whose value is a (linear) function of a basic induction variable, e.g., the address expression of $A[i]$ in Figure 8 (a).

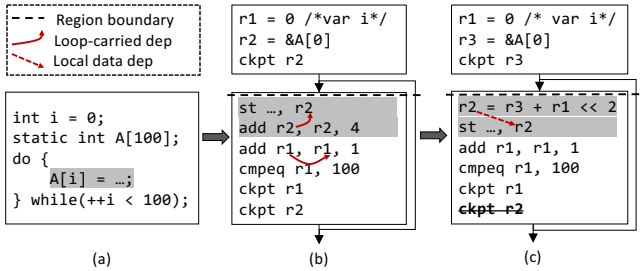

Figure 8: (a) original C code, (b) strength reduction code, and (c) LIVM enabled code eliminating $r2$'s checkpoint

Here, the compiler's loop strength reduction [46] turns the expression $(\&A[0] + i * 4)$ into a separate basic induction variable that is initialized as $\&A[0]$ and increased by 4 as shown in Figure 8 (b). The problem is that the strength reduction results in loop-carried data dependence, i.e., $r2$ is used in the next iteration as the address operand of a store, rendering $r2$ live-out [2] and checkpointed in the loop thus degrading the performance; Figure 8 (b) highlights the strength reduction enabled code in the shaded box and shows the resulting checkpoint, i.e., *ckpt* $r2$, in the bottom.

To deal with the problem, Turnpike proposes a new optimization called loop induction variable merging (LIVM). It investigates basic induction variables in a loop to see if one can be merged to some other basic induction variable in form of an expression derived from the basic induction variable. In other words, LIVM makes the merged variable become an induced induction variable so that it can eliminate the loop-carried data dependence. As shown in Figure 8 (c), $r2$ has only local data dependence with the help of LIVM; since $r2$ is no longer live-out, Turnpike eliminates the *ckpt* $r2$ in the bottom of the figure. Since it used to be executed every loop iteration, the impact of its elimination on the store buffer traffic reduction should be very significant if enabled.

4.1.3 Optimal Checkpoint Pruning. To further reduces checkpoint stores, Turnpike leverages optimal checkpoint pruning [29] in the recent advance of GPU register file protection called Penny. We found the pruning technique effective for reducing the store buffer pressure, though it is originally devised for GPUs—that have never had a store buffer (SB)—and idempotent regions [29] that are intrinsically different from Turnpike's SB-size aware partitioned regions.

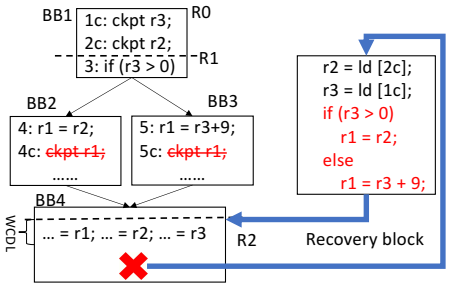

Figure 9: Checkpoint pruning for eliminating $4c$ and $5c$

[2] A region boundary is placed in a loop header as in Turnstile [39].

It turns out that Penny's checkpoint pruning removes a large number of checkpoints without compromising the recoverability guarantee. The key idea is that it is safe to remove those checkpoints, provided the value to be checkpointed can be reconstructed from a constant or the value of other checkpoints at the recovery time of an error detected. Turnpike exploits Penny's optimal pruning algorithm that can detect unnecessary checkpoints in polynomial time with the recovery code generated to reconstruct the pruned checkpoint value.

Figure 9 shows how the checkpoint pruning works and ensures safe recovery. Suppose an error is detected in a region R2 in the bottom of the figure; R2's input registers $r1, r2, r3$ have been checkpointed by prior regions, e.g., the first region R0 checkpoints $r2$ and $r3$. Similarly, without the checkpoint pruning, $r1$ would be checkpointed by the middle region R1 where either $r2$ or $r3$ is used to update $r1$ depending on the path taken in the branch. Here, either way, $r1$'s checkpoint ($4c$ and $5c$) can be removed since it can be reconstructed by using the checkpointed value of $r2$ or $r3$ in the recovery block. To recover from the error here, the recovery block of the region R2—starting from the recovery PC (Section 2)—executes the backward slice of the pruned checkpoint, which includes the branch to reconstruct $r1$ differently according to the checkpointed predicate $r3$, and jumps back to the recovery PC, i.e., the beginning of the region R2.

4.1.4 Moving a Checkpoint out of a Loop with LICM. Although the pruning scheme investigates if a checkpoint can be safely eliminated, it never tries to move the location of the checkpoint. For those checkpoints that cannot be eliminated, the pruning scheme leaves them at their original checkpointing location, thus losing a chance to move a checkpoint out of the loop body. The main reason for this is that under the eager checkpointing policy, a checkpoint cannot but be placed right after the instruction that updates the live-out register in each region.

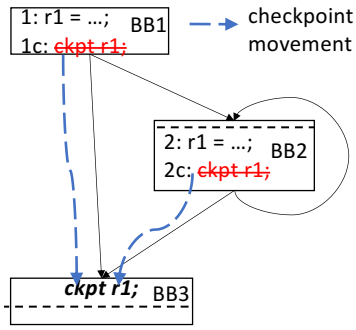

Figure 10: LICM at $2c$

Interestingly, the eager checkpointing can be relaxed without compromising the recoverability. Recall that a checkpoint is necessary for 2 reasons: (1) saving the registers that are input to some later regions and (2) verifying the integrity of the register value, i.e., no register corruption. In the input-saving point of view alone, Turnpike only has to checkpoint the register before it is used. On the other hand, to verify the register, it must be saved before the region is finished due to the region-level verification (Section 2.1). As a result, for each checkpoint in a given region, the checkpoint can be safely moved from the original eager checkpointing location down to any points before the region boundary.

Figure 10 shows how Turnpike leverages this insight to move $r1$'s checkpoint at line $2c$ out of a loop as in LICM (loop invariant

code motion[3]). By moving the checkpoint down to near the region boundary below, Turnpike takes the checkpoint off from the loop body. Moreover, since the checkpoint is now placed in the bottom basic block, another checkpoint at line 1*c* becomes redundant—as they both checkpoint the same value of *r*1—and can be safely eliminated as well. That way Turnpike can reduce the performance overhead significantly for some applications (Section 6.3).

4.2 Hiding the Execution Delay of Checkpoints

There are still many remaining checkpoints that cannot be removed by the prior 2 optimizations. Since Turnstile inserts each checkpoint store right after the register-update instruction, the store's dependence on the register (data hazard) often makes the in-order pipeline stall till the register gets ready (see Section 3.3). To address this problem, Turnpike leverages another compiler optimization, i.e., instruction scheduling [1, 11, 46]. It attempts to separate the register-update instruction from the dependent checkpoint store instruction by hoisting some of the following independent instructions beyond the store instruction.

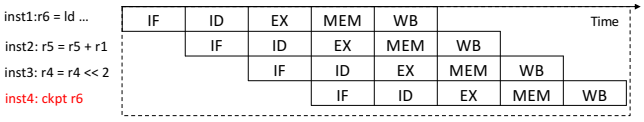

Figure 11: Execution delay of checkpoint gets reduced by rescheduling instruction stream

Figure 11 shows how the instruction scheduling can handle the checkpoint data hazard in Figure 6. With the scheduling, the checkpoint store for a register *r*6 being loaded is moved down in Figure 11; that way the store can be executed with no stall, i.e., its operand *r*6 is ready from the load—because the load latency is overlapped with the execution of 2 other intervening instructions before the store. Since the register becomes available when the reordered store is about to execute, it can avoid the data hazard. Note that the instruction scheduling helps the pressure on the store buffer (SB) to be relieved as well. The reason is that the reordered stores can eventually reduce the execution time of their region, which is the part of the region verification time; it consists of the region execution time and the WCDL as shown in Figure 1(a). Hence, this also reduces the time during which stores stay in the SB for verification, keeping the pressure for a shorter amount of time.

4.3 Relieving the Store Buffer Pressure

Unlike other optimizations, the next two novel hardware schemes in this section can directly relieve the store buffer pressure. Turnpike achieves that by releasing some of the buffered stores to cache without verification yet in a manner that still guarantees the soft error resilience. To begin with, we classify store instructions into two kinds: (1) regular stores stemming from the program itself or register allocation, i.e., spill stores to stack, and (2) checkpoint stores generated to save updated live-out registers. The rest of this section discusses the two kinds and how they are addressed by our 2 hardware schemes, respectively.

[3] While LICM is to hoist the invariant code out of a loop [1, 11], most of the production compilers (GCC/LLVM) have extended LICM to support code sinking too. We modified the LLVM passes to move down checkpoints in a loop as they are guaranteed not to be aliased with other memory operations.

4.3.1 Fast Release of Regular Stores. Prior work, Turnstile, has all stores of a region quarantined in a store buffer (SB) till the region turns out to be error-free for both region verification and in-core error containment purposes (Section 2). However, we found out that not all the data being stored are going to be read for the verification of a region. For example, the data stored at line 3 in Figure 12 is never read in the region R1 when R1 is restarted upon an error due to the absence of write-after-read (WAR) dependence; we refer to such a store as a WAR-free store. Thus, even if the data is corrupted due to an error and written to cache, R1's re-execution can correctly recover from the error. With that in mind, Turnpike releases such a WAR-free store—without verification—immediately after its commit, thereby relieving the store buffer pressure.

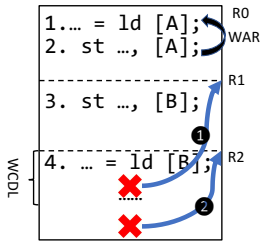

Figure 12: Fast release of a WAR-free regular store

One might suspect that due to the fast release of unverified data to cache, the next region might read it making the error recovery fail, e.g., data stored at line 3 by a region R1 is loaded at line 4 by the following region R2 in Figure 12. Fortunately, it turns out that this is not a problem at all. For the unverified yet corrupted data to be read by the region R2's load, the error must be detected before R1's verification point, i.e., within WCDL (e.g., 10) cycles after the prior region R1 is finished. However, it is not R2 but R1 that the original region-level soft error verification (Section 2) restarts to recover from the error (❶ in Figure 12). Again, R1 does not read the data, i.e., no WAR dependence, and therefore restarting R1 can correct the error with no harm. On the other hand, if the error is detected after R1's verification point, then R2 is restarted for recovery (❷ in Figure 12). In this case, R2's load is guaranteed to read the correct data—because it was written by the region R1 which has already been verified.

The takeaway is that WAR-free stores can bypass the verification and thus can be immediately merged to cache after their commit, whether the error is detected during the execution of their region or within WCDL cycles after the region is finished. To realize this, Turnpike proposes a novel microarchitectural technique called committed load queue (CLQ)—shown in Figure 2—to dynamically check the absence of WAR dependence for each regular store.

Ideal CLQ Design with Address Matching. For each committed load, Turnpike allocates an entry in the CLQ to keep the address of the load. When the in-order pipeline tries to commit a regular store, Turnpike compares its address to all the entries of CLQ to check whether the store has WAR dependence on any prior load in the current region. If there is no address conflict, i.e., no WAR dependence, Turnpike releases the WAR-free store immediately instead of holding it in the store buffer. Otherwise, it is quarantined in the store buffer as is for the original region-level verification.

Once each region gets verified, Turnpike clears only the CLQ entries that were populated during the execution of the region. In particular, if the CLQ is full, Turnpike does not stall the pipeline. Whenever CLQ overflows, it instead disables the fast release logic for WAR-free stores, i.e., load address insertions to CLQ are blocked and it is wiped out, making the following stores go through the SB quarantine as is. When a new region starts thereafter, Turnpike resumes the CLQ insertion so that the region can leverage the fast release of its WAR-free stores unless the CLQ overflows. More precisely, to ensure in-order store release to L1 cache, Turnpike does not enable the fast release logic until the prior region is verified with its stored released. Figure 13 illustrates how Turnpike selectively controls (enables/disables) the fast release of WAR-free stores according to the CLQ status, i.e., whether it is full or not.

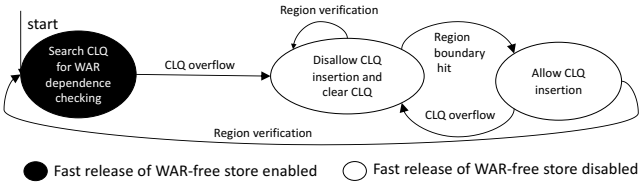

Figure 13: Selective control for WAR-free stores' fast release

Compact CLQ Design with Range Checking. In general, the bigger CLQ size is, the more often the fast release logic is enabled—leading to more WAR-free stores that can be merged to cache without their region verification. However, we found out that the WAR dependence is rarely found in each region. Taking this into account, we propose a range-based address checking that can compress all the addresses of the loads executed in each region by keeping the range of the minimum and maximum addresses during the execution. In this way, Turnpike only needs to allocate a single CLQ entry for each region without hurting the precision significantly.

Furthermore, such a per-region range-based CLQ entry renders the WAR dependence checking logic faster and simpler, which would otherwise require CAM (content-addressed memory) search for multiple entries. That is, for each regular store of a given region, Turnpike (1) looks up the CLQ entry corresponding to the region and (2) checks if the store address falls into the address range of the entry. The upshot is that Turnpike can significantly reduce the hardware cost for CLQ with neither the significant loss of the precision to detect WAR-free stores nor the visible performance degradation compared to the address matching based ideal CLQ.

To confirm this, we compare the performance of Turnpike's compact CLQ against the ideal (100%-accurate) CLQ that performs address matching to identify WAR-free stores with an infinite number of CLQ entries. Figure 14 shows the performance overhead of the 2 designs which is normalized to the original application execution time that has no soft error resilience. It turns out that Turnpike's compact CLQ design only incurs 3% performance loss on average compared to the infinite-size ideal CLQ. As shown in Figure 15, that is because the infinite-size ideal CLQ leads to 10.58% higher detection accuracy than Turnpike's compact CLQ.

Finally, since Turnpike's compact CLQ has only 2 entries by default, it is technically possible to encounter the CLQ overflow,

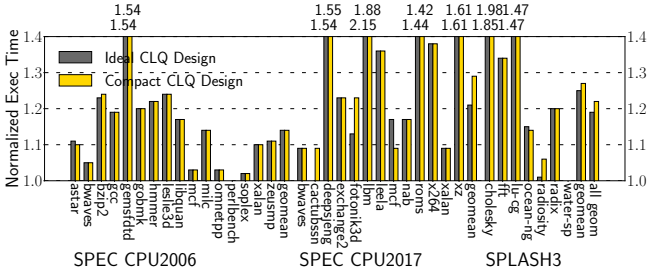

Figure 14: Run-time overhead compared to original program without resilience support between an ideal CLQ (infinite-size CLQ) and Turnpike's compact 2-entry CLQ; note that we only enable WAR-free checking and hardware coloring to exclude the impacts of Turnpike compiler optimizations

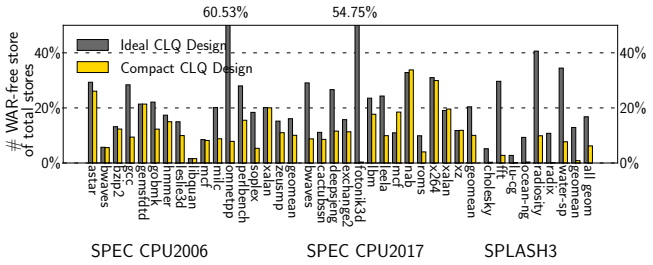

Figure 15: Ratio of detected WAR-free stores to all stores including checkpoints (higher is better) for the ideal basic CLQ (infinite-size) and Turnpike's compact CLQ (2 entries)

that is handled by the selective fast release control (Figure 13). For example, suppose that Turnpike executes three consecutive regions; while the first region is being verified, Turnpike can reach the end of the second region in which case CLQ does not have an available entry—due to the overflow—for accommodating the addresses of the last region's loads. In fact, Turnpike's compiler ensures that each region cannot have more than half of the SB size so that the verification of one region can be overlapped with the execution of the next region. However, since the region partitioning [39] is based on a path-insensitive analysis [46], some regions might have even a smaller number of stores than the half of the SB size, e.g., regions could have only one store when the SB has 4 entries as with ARM Cortex A53. As will be shown in Figure 24, the compact CLQ needs 3-4 entries to prevent the overflow for all our benchmarks.

4.3.2 Fast Release of Checkpoint Stores. By definition, all checkpoints are a WAR-free store in their region because the register stored by a checkpoint is never read by its own region; rather, the register is only used as an input to some later region. For this reason, one might think it is ok to release checkpoint stores without the SB quarantined for verification. However, we found it impossible because the error recovery could fail sometimes.

Figure 16 describes such a corner case with two regions R0 and R1 that both checkpoint the same register *r*2. Suppose *r*2 at line 2 is corrupted due to a soft error, and the error is detected after the verification point of the prior region R0. That is, the next region R1 is to be re-executed for the error recovery. Here, if a checkpoint of *r*2 at line 2c is merged to cache without verification—though

it is corrupted, the checkpoint memory location is going to be overwritten by the corrupted value of $r2$. Hence, the re-execution of R1 ends up restoring its input register $r2$ from the corrupted value, thereby failing to correct the error.

The crux of the problem is that the checkpoint storage (location) is overwritten. With that in mind, we prevent the overwriting with alternative storage. This allows Turnpike to safely release even checkpoint stores immediately after their commit bypassing the verification. To achieve this, Turnpike leverages simple microarchitectural support called hardware coloring that can dynamically assign a distinct memory location (i.e., color) to a checkpoint. As such, Turnpike prepares a coloring pool, i.e., a set of memory locations as checkpoint storages, to manage the available colors for each register.

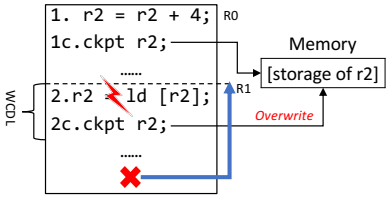

Figure 16: Problem of releasing of checkpoints without verification

Implementation Details of Hardware Coloring. To ensure that a distinct color is assigned to each checkpoint, Turnpike prepares a 4-color pool, i.e., there are 4 checkpoint memory locations for each register, and manages 3 register maps: Available_Colors (AC), Used_Colors (UC) and Verified_Colors (VC). For a given register, AC maps it to the next available color while UC to the color that is used (assigned) for each region; Turnpike maintains UC as a part of RBB entry as shown in Figure 2. Likewise, VC maps a given register to the verified color (the checkpoint storage)—from which Turnpike restores the input register of the region being restarted on recovery.

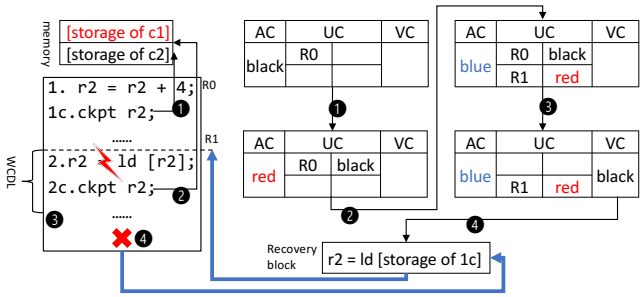

Figure 17: Fast release of a checkpoint store

Initially, VC is empty since there is nothing verified. When the in-order pipeline encounters a checkpoint, Turnpike tries to assign a color to the checkpoint by referring to AC with the register being checkpointed. If there is an available color, it is inserted to the UC of the region in which the checkpoint store exists; otherwise, Turnpike simply gives up the fast release of the checkpoint store and falls back to the store buffer quarantine for verification.

Figure 17 shows how the status of AC, UC, and VC changes for register $r2$ being checkpointed. Here, checkpoint at line 1c is assigned **black** from AC, and thus the UC of a region R0 is updated with **black** (❶ in the figure). Similarly another checkpoint 2c is assigned **red**, and the corresponding register mapping in the UC of region R1 is updated with **red** (❷).

Once a region is verified, Turnpike flushes every color of VC to AC for reclamation purpose and updates the VC with the used colors of the verified region which are obtained by searching the UC with the region as a key. As shown in Figure 17, once R0 is verified at the end of WCDL after it is finished, Turnpike updates the VC with the color(s) that UC holds for R0, i.e., **black** (❸).

When an error is detected in a region R1 at some point after WCDL (❹) in the figure, Turnpike invokes the recovery block to restore the value of the input register $r2$ from the **black** checkpoint storage, and then jumps back to the recovery PC, i.e., the entry of R1. Overall, Turnpike's hardware cost is not significant as will be shown in Section 6.5.

5 DISCUSSION

Fault Model: We assume that both SB and RBB are hardened to be robust against soft errors as in prior work [39]. Besides, the 2 entries of the committed load queue (CLQ) and the three color maps (total 6 bits per register) are to be protected. Like prior work and commodity RAS (reliability/availability/serviceability) processors, caches and the address generation unit (AGU) should be hardened. Finally, a single parity bit is necessary for each register in case it holds store's address operand whose corruption ends up altering random memory location under Turnpike's fast release. Turnpike prevents this problem by causing any parity-detected error upon each register access to trigger its recovery process—as if it were detected by acoustic sensors.

Store Buffer Scaling: In general, it is challenging to enlarge a store buffer because SB's store-to-load forwarding impacts the length of a pipeline clock tick. SB must provide data within L1-hit time to avoid complications of scheduling loads with variable latency, e.g., for a 16/32-entries SB in Alpha AXP processor clocked at 3GHz, the store-to-load forwarding latency increases to 3-4 cycles[59]. Especially for in-order cores, it is even more challenging due to the power-hungry nature of the CAM (content-addressed memory) search for the store-to-load forwarding. That is why commodity in-order cores have only a few SB entries, e.g., ARM Cortex-A53 has 4 entries. In addition, Section 6.5 discusses the area and energy overheads of a large SB design in detail.

6 IMPLEMENTATION AND EVALUATION

6.1 Methodology

We implemented our optimizations presented in Section 4 with LLVM compiler [32]. To evaluate Turnpike's performance, we used SPEC2006[23]/SPEC2017[7] and SPLASH3[56] compiling all benchmarks with -O3. We conducted simulation using gem5 [6] which is configured with 2-issue, 2.5GHz dual-core processor with 32KB/64KB 2-way set-associative L1 instruction/data caches (2 cycles hit) and a unified 128KB 16-way set-associative L2 cache (20 cycles hit) to model an ARM Cortex-A53 processor [2]. The store buffer size is set to 4 as with the recent work that simulates the Cortex-A53 core [28], and the default CLQ size is 2. According to prior works [67–70], 300-30 deployed acoustic sensors can achieve 10-30 cycles of the

worst-case detection latency (WCDL) with the area cost of less than 1% of die size, and therefore we set the default WCDL to 10 cycles.

For SPEC CPU2006 and SPEC CPU2017, we synchronized the number of simulated instructions by measuring the number of the function call instructions which is a constant across binary versions generated by different compiler optimizations. All benchmarks were fast-forwarded through the number of function calls to execute at least 5 billion instructions on the original executable without soft error resilience support, then we simulated the next 1 billion instructions with the gem5 in-order pipelined processor model. To be more practical, we simulated all SPEC CPU benchmarks with the reference inputs. For SPLASH3 benchmarks, we simulated the entire program with full system model of gem5. In the following, all performance results are presented as a slowdown, i.e., the inverse of a speedup, to the baseline that has no soft error resilience support.

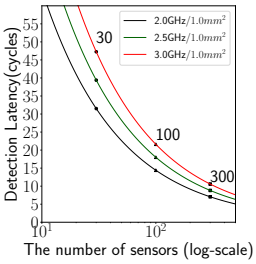

Figure 18: Detection latency across the number of deployed sensors

6.2 Run-time Overhead with Varying WCDL

WCDL (worst-case detection latency) is inversely proportional to the number of sensors deployed and affected by the underlying clock frequency; the higher frequency the clock is, the longer the WCDL is. Figure 18 shows these trends for 300-30 sensors deployed on top of $1mm^2$ core die, e.g., 10 cycles WCDL for 2.5GHz core with 300 sensors. Due to the process technology and the fabrication issue, deploying all 300 sensors might not be possible under the budget of 1% die size overhead. Thus, we vary WCDL from 10 cycles up to 50 cycles to cover other possible fabrication cases and evaluate the general trend of Turnpike's overhead across the different WCDLs.

Figure 19 presents the run-time overhead of Turnpike for 5 WCDLs: 10/20/30/40/50. Turnpike incurs only 0-14% overheads on average for the varying WCDLs from 10 to 50 cycles. In contrast, Turnstile suffers 29-84% average overheads for the 5 WCDLs (see Figure 20). It is worth noting that Turnpike significantly outperforms Turnstile for all the benchmarks. In particular, when 10-cycle WCDL is used by default, Turnpike's overhead is only around 1% for most of the benchmarks, thereby delivering 0% average overhead!

6.3 Impact of Turnpike's Optimizations

This section presents the performance impact of Turnpike's optimizations. Figure 21 shows the performance results of the following 8 cases for the default 10-cycle WCDL.

Turnstile: is the state-of-the-art work, that does not use our optimizations, and incurs a 29% overhead on average.

WAR-free Checking: uses the fast release of regular stores; it reduces the overhead to 25%.

Fast Release (WAR-free checking and HW coloring): enables the fast release of both regular stores and checkpoint stores; this reduces the overhead to 22%.

Fast Release + Pruning: is the combination of the above fast release and checkpoint pruning, achieving 12% overhead; the latter removes many unnecessary checkpoints (Figure 23).

Fast release + Pruning + LICM: is the combination of the fast release, checkpoint pruning, and LICM, achieving 10% overhead. LICM particularly works well for `deepsjeng`, `fotonik3d`, `nab`, and `x264`, reducing their overhead by >5%.

Fast release + Pruning + LICM + Inst Sched: is the combination of the fast release, checkpoint pruning, LICM, and instruction scheduling. The resulting average overhead is 7%.

Fast release + Pruning + LICM + Inst Sched + RA Trick: is the combination of the fast release, checkpoint pruning, LICM, instruction scheduling, and store-aware register allocation. On average, it reduces the overhead to 2%. Significant overhead reduction is found in `gemsfdtd` and `lbm`; as shown by Figure 23, the register

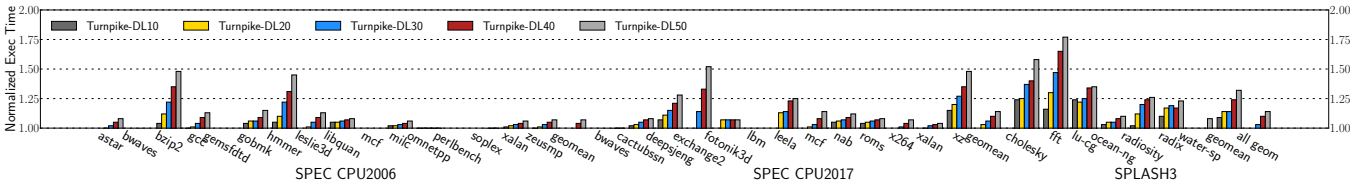

Figure 19: Performance overhead of Turnpike with varying WCDL from 10 to 50 cycles

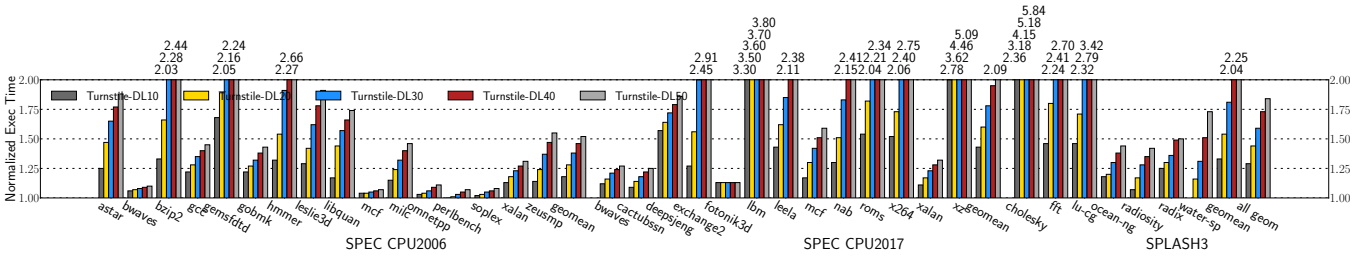

Figure 20: Performance overhead of Turnstile with varying WCDL from 10 to 50 cycles

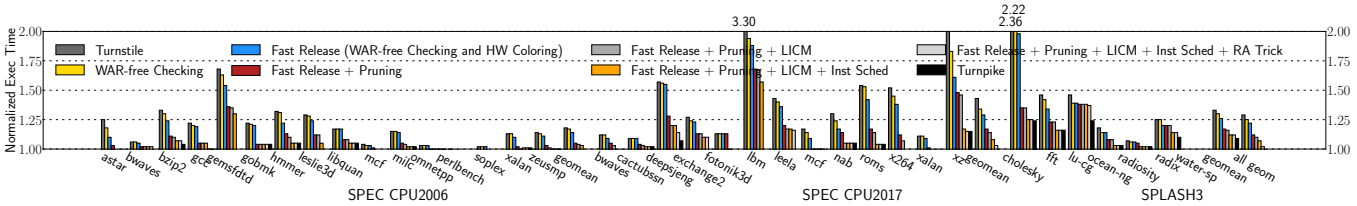

Figure 21: Performance comparison between Turnstile and Turnpike's optimizations with 10-cycle WCDL.

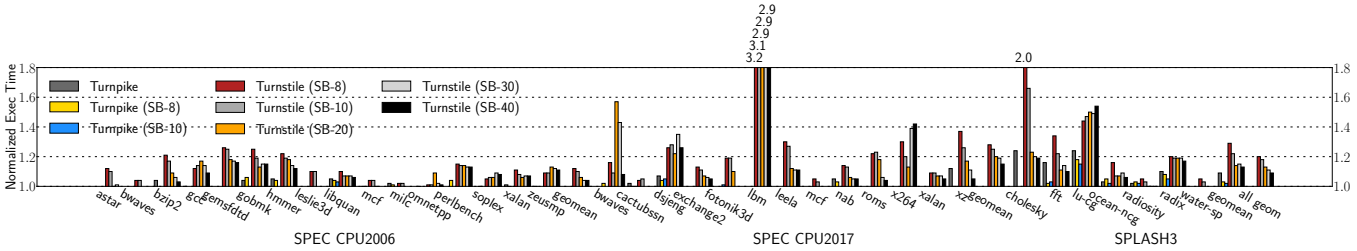

Figure 22: Performance comparison of Turnpike and Turnstile with different SB sizes (8, 10, 20, 30, 40) using 10-cycle WCDL

allocation trick eliminates the stores of the 2 benchmarks by 19% and 17%, respectively.

Turnpike: uses all above optimizations along with loop induction variable merging, eliminating Turnstile's overhead completely, i.e., Turnpike's average overhead is 0%! It turns out that loop induction variable merging is particularly effective for `exchange2`, `leela`, `lu-contigous`, and `radix`.

	Area (μm^2)	Dynamic access (pJ)
4-entry SB (CAM)	621.28	0.43099
Color maps in Turnpike (RAM)	36.651	0.02518
2-entry CLQ in Turnpike (RAM)	24.434	0.01679
Turnpike in total (color maps + 2-entry CLQ)	61.085	0.04197
40-entry SB (CAM)	3132.50	2.11525
Turnpike in total / 4-entry SB	9.8%	9.7%
40-entry SB / 4-entry SB	504%	497%

Table 1: Cost comparison of Turnpike and a large SB design

6.4 Impact of SB Pressure Reduction Schemes

Figure 23 shows the detailed breakdown of all stores with 8 categories: **Pruned** corresponds to the checkpoint stores eliminated by the optimal checkpoint pruning while **LICM-eliminated** to those removed by the loop-invariant code motion. Among the remaining checkpoint stores, **Colored** corresponds to those that can be merged to cache without the SB quarantine. Similarly, **WAR-free** corresponds to the regular stores that can be merged to cache without verification. Next, **RA-eliminated** and **IndVarMerging-eliminated** correspond to those stores that can be removed by our store-aware register allocation and loop induction variable merging optimization respectively. Finally, **Others** represents the rest of the stores which cannot be removed or fast released by Turnpike thus going through the verification. As shown in the figure, the checkpoint pruning removes 21% of all stores while LICM removes 1.4% of them on average. Although LICM has little impact for the majority of the benchmarks, its checkpoint removal is significant for `cactubssn`, `lbm`, `cholesky` and `radix`. Meanwhile, 1.7% and 5% of all stores are removed by store-aware register allocation and loop induction variable merging respectively. Finally, 39% of all stores can be released to cache without going through the SB quarantine, highlighting the effectiveness of Turnpike's fast release.

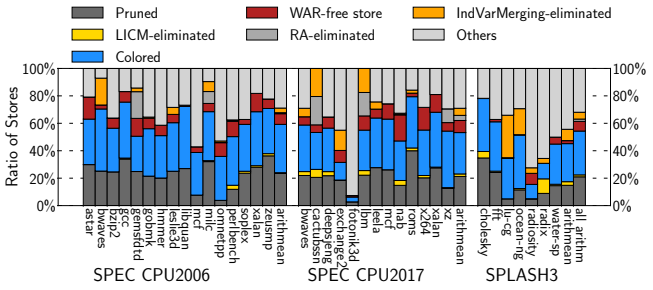

Figure 23: Store breakdown; 2-entry CLQ; 10-cycle WCDL

6.5 Hardware Cost Analysis

Turnpike incurs a very small hardware overhead; the 2-entry CLQ requires 16 bytes while the three 4-color maps (AC, UC, and VC in Section 4.3.2) need 6 bits ($3 \cdot \log_2 4$) per register—requiring 24 bytes for 32 registers as in ARM Cortex A53. In summary, Turnpike only needs total 40 bytes for such an in-order processor.

To further evaluate the area and the power overheads of Turnpike, we used CACTI [60] with 22*nm* technology. Table 1 highlights the area/power-efficiency of Turnpike's compiler/architecture co-design. Compared to ARM Cortex A53's 4-entry store buffer as a baseline, Turnpike only incurs 9.8% area and 9.7% energy overheads (see the second last low of the table). In contrast, simply increasing the store buffer size to 40 causes 504%/497% area/energy overheads, which is unrealistic for low-power in-order cores.

6.6 Sensitivity Analysis

Sensitivity to CLQ size: Recall that CLQ is a critical hardware structure, since its dependence checking logic is essential for the fast release of WAR-free stores. Figure 24 shows the average and

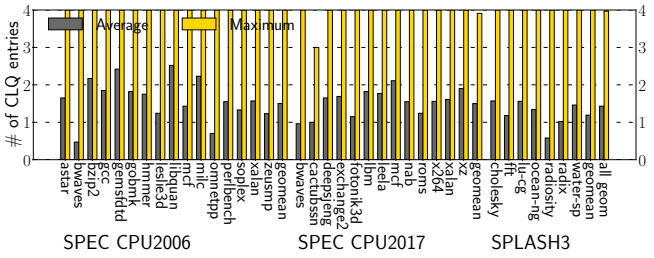

Figure 24: Dynamic CLQ entries populated; 10-cycle WCDL

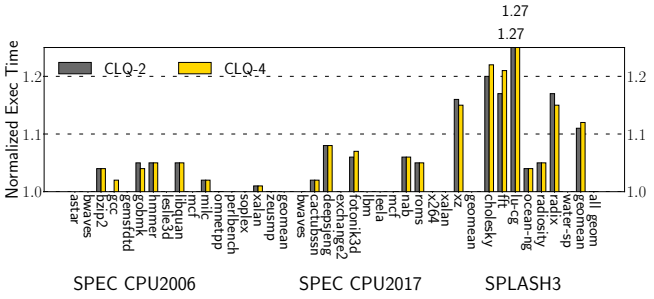

Figure 25: 2-entry vs 4-entry CLQs with 10-cycle WCDL

maximum numbers of dynamic CLQ entries populated at run time. The average number of populated CLQ entries is about 1 though the maximum number goes up to 3 or 4 for some applications. Further investigation confirms that the peak number is scarcely observed. That is why Turnpike's CLQ size is set to 2 (by default), and its performance is almost the same as that of a bigger CLQ with 4 entries as shown in Figure 25. The takeaway is that our compact CLQ design is not only low-cost but also high-performance.

Sensitivity to SB Size: It is hard to increase the size of a store buffer (SB) especially for in-order cores. Nonetheless, to highlight the performance of Turnpike, we enlarge the store buffer of Turnstile—though it performs poorly for the 4-entry SB of ARM Cortex A53 which is Turnpike's SB size. In addition to the default size, we tested 5 more SB sizes from 8 to 40 with 10-cycle WCDL. As shown in Figure 22, for 5 SB sizes (8/10/20/30/40) with 10-cycle WCDL, Turnstile's average performance overheads are 20%, 18%, 13%, 11%, and 9%, respectively. Note that although Turnstile is equipped with a much larger SB, it performs significantly worse than Turnpike. Even with the 40-entry SB that is 10x bigger than Turnpike's SB, the average slowdown of Turnstile is 9% whereas that of Turnpike is 0% (see Figure 21). We also tested Turnpike for bigger SB sizes. Figure 22 shows that the average overhead of Turnpike is still 0% with the SB sizes of 8 and 10 and decreases as the SB size increases.

6.7 Region Size and Code Size Analysis

Figure 26 shows dynamic region size and binary code size increase. On average, there are 11.2 instructions per region, and code size increases by 0.4% compared to the baseline. Overall, long regions lead to less code size increase; bwaves has 35 instructions per region leading to 0.35% increase, while gcc increases the size by 8.15% due to many small regions (7.8 instructions per region).

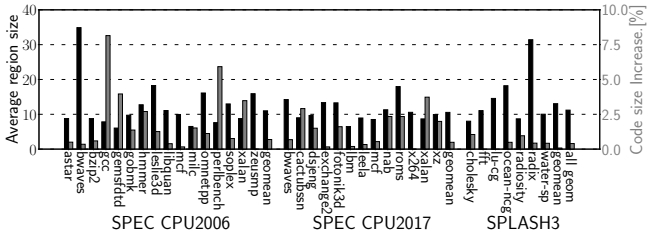

Figure 26: Region size (left) and binary overhead (right bar)

7 OTHER RELATED WORKS

Many prior works use redundant-computation-based detection for high error coverage. Instruction-level duplication replicates instructions and detects errors by comparing the results of the original and replica instructions [16, 17, 17, 31, 41, 44, 52, 54]. In contrast, redundant multithreading simultaneously runs a redundant thread with the original thread on available cores. Some schemes use SW techniques to realize the redundant multithreading without hardware modification [45, 63, 64, 74, 77], while others exploit HW support to reduce the performance overhead [30, 48, 53, 55, 62, 74], Another schemes with process-level redundancy duplicate the application process to compare the outputs [18, 61, 76] between the parent and child processes. Finally, non-duplication schemes detects errors by catching abnormal symptoms caused by a soft error [20, 22, 33, 75].

To recover from detected errors, triple module redundancy (TMR) adopts a majority voting between the 3 executions, increasing the hardware cost. The most common error recovery scheme is to use checkpointing or logging program status (register and memory). Prior work on coarse-grained recovery requires expensive hardware support for incrementally checkpointing memory status [69] or equipping the core with a large store buffer for memory logging [65]. To reduce the checkpointing cost in a fine-grained manner, recent studies partition the program into small idempotent regions to reduce the number of data to be checkpointed [13, 14, 35, 38]. However, the idempotent recovery schemes still incur a significant run-time overhead due to register spilling or checkpointing.

All prior works either impose a large hardware cost or suffer a high run-time cost. To the best of our knowledge, Turnpike is the first, that reduces both costs effectively for in-order cores, requiring little hardware cost though it achieves almost 0% run-time overhead.

8 CONCLUSION

This paper presents Turnpike that achieves lightweight soft error resilience for in-order cores with acoustic-sensor-based detection. Using compiler optimizations and simple hardware support, Turnpike incurs near-zero performance overhead.

ACKNOWLEDGMENTS

At Purdue, this work was supported by NSF grants 1750503 and 1814430. At SNU, this work was supported in part by the National Research Foundation of Korea (NRF) grants (NRF-2016M3C4A7952587 and NRF-2019M3E4A1080386) and by the Institute for Information & communications Technology Promotion grant (No. 2018-0-00581, CUDA Programming Environment for FPGA Clusters), all funded by the Ministry of Science and ICT of Korea. ICT at SNU provided research facilities for this study.

REFERENCES

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

[2] ARM. [n.d.]. ARM Cortex-A53 Processor Technique Reference Manual. http://infocenter.arm.com/help/topic/com.arm.doc.ddi0500d/DDI0500D_cortex_a53_r0p2_trm.pdf

[3] ARM. 2018. The Arm Automotive Guide: Arm-based Automotive Partner Demonstrations Highlights for CES 2020.

[4] ARM. 2018. How to Make Autonomous Vehicles a Reality with Arm.

[5] Karl Berntorp, Pranav Inani, Rien Quirynen, and Stefano Di Cairano. 2019. Motion planning of autonomous road vehicles by particle filtering: Implementation and validation. In *2019 American Control Conference (ACC)*. IEEE, 1382–1387.

[6] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. 2011. The Gem5 Simulator. *SIGARCH Comput. Archit. News* 39, 2 (Aug. 2011), 1–7. https://doi.org/10.1145/2024716.2024718

[7] 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.

[8] Chao Chen. 2020. *Compiler-Assisted Resilience Framework for Recovery from Transient Faults.* Ph.D. Dissertation. Georgia Institute of Technology.

[9] 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.

[10] Reece A Clothier, Dominique A Greer, Duncan G Greer, and Amisha M Mehta. 2015. Risk perception and the public acceptance of drones. *Risk analysis* 35, 6 (2015), 1167–1183.

[11] Keith Cooper and Linda Torczon. 2011. *Engineering a compiler*. Elsevier.

[12] XTIM Corporation. 2021. Bionic Bird (Biomimetic Drone) Instruction Manual. https://bionicbird.com/wp-content/uploads/pdf/Bionicbird_manual/Bionic_Bird_Deluxe_Package_Manual_EN.pdf.

[13] 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.

[14] Marc A De Kruijf, Karthikeyan Sankaralingam, and Somesh Jha. 2012. Static analysis and compiler design for idempotent processing. In *ACM SIGPLAN Notices*, Vol. 47. ACM, 475–486.

[15] N DeBardeleben, S Blanchard, V Sridharan, S Gurumurthi, J Stearley, K Ferreira, and J Shalf. 2014. Extra Bits on SRAM and DRAM Errors - More Data From the Field. *Silicon Errors in Logic - System Effects (SELSE-10), Stanford University* (April 1, 2014 2014).

[16] 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.

[17] 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.

[18] Björn Döbel and Hermann Härtig. 2014. Can we put concurrency back into redundant multithreading?. In *Proceedings of the 14th International Conference on Embedded Software*. 1–10.

[19] Guy Durrieu, Madeleine Faugère, Sylvain Girbal, Daniel Gracia Pérez, Claire Pagetti, and Wolfgang Puffitsch. 2014. Predictable flight management system implementation on a multicore processor. In *Embedded Real Time Software (ERTS'14)*.

[20] Shuguang Feng, Shantanu Gupta, Amin Ansari, and Scott Mahlke. 2010. Shoestring: probabilistic soft error reliability on the cheap. *ACM SIGARCH Computer Architecture News* 38, 1 (2010), 385–396.

[21] Sebastian Hahn and Jan Reineke. 2019. Design and analysis of SIC: A provably timing-predictable pipelined processor core. *Real-Time Systems* (2019), 1–39.

[22] Siva Kumar Sastry Hari. 2009. *Low-Cost Hardware Fault Detection and Diagnosis for Multicore Systems running Multithreaded Workloads.* Ph.D. Dissertation. University of Illinois at Urbana-Champaign.

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

[24] Carles Hernandez and Jaume Abella. 2015. Timely error detection for effective recovery in light-lockstep automotive systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 34, 11 (2015), 1718–1729.

[25] Texas Instruments. 2018. Advanced Driver Assistance (ADAS) Solutions Guide.

[26] Xabier Iturbe, Balaji Venu, Emre Ozer, Jean-Luc Poupat, Gregoire Gimenez, and Hans-Ulrich Zurek. 2019. The Arm Triple Core Lock-Step (TCLS) Processor. *ACM Transactions on Computer Systems (TOCS)* 36, 3 (2019), 1–30.

[27] Wonjin Jang. 2011. *Soft-error tolerant quasi delay-insensitive circuits.* Ph.D. Dissertation. California Institute of Technology, Pasadena, CA, USA.

[28] Ipoom Jeong, Seihoon Park, Changmin Lee, and Won Woo Ro. 2020. CASINO Core Microarchitecture: Generating Out-of-Order Schedules Using Cascaded In-Order Scheduling Windows. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 383–396.

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] Man-Lap Li, Pradeep Ramachandran, Swarup K Sahoo, Sarita V Adve, Vikram S Adve, and Yuanyuan Zhou. 2008. Swat: An error resilient system. *Proceedings of SELSE* (2008).

[34] Qingrui Liu. 2018. *Compiler-directed error resilience for reliable computing.* Ph.D. Dissertation. Virginia Tech.

[35] 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.

[36] 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.

[37] Qingrui Liu, Changhee Jung, Dongyoon Lee, and Devesh Tiwari. 2015. Clover: Compiler directed lightweight soft error resilience. In *ACM Sigplan Notices*, Vol. 50. ACM, 2.

[38] 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.

[39] 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.

[40] 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.

[41] Junchi Ma and Yun Wang. 2016. Identification of Critical Variables for Soft Error Detection. In *International Conference on Human Centered Computing*. Springer, 310–321.

[42] Aakar Mehra, Wen-Loong Ma, Forrest Berg, Paulo Tabuada, Jessy W Grizzle, and Aaron D Ames. 2015. Adaptive cruise control: Experimental validation of advanced controllers on scale-model cars. In *2015 American Control Conference (ACC)*. IEEE, 1411–1418.

[43] Jörg Mische, Irakli Guliashvili, Sascha Uhrig, and Theo Ungerer. 2010. How to enhance a superscalar processor to provide hard real-time capable in-order smt. In *International Conference on Architecture of Computing Systems*. Springer, 2–14.

[44] Konstantina Mitropoulou, Vasileios Porpodas, and Marcelo Cintra. 2013. DRIFT: Decoupled compiler-based instruction-level fault-tolerance. In *International Workshop on Languages and Compilers for Parallel Computing*. Springer, 217–233.

[45] Konstantina Mitropoulou, Vasileios Porpodas, and Timothy M Jones. 2016. COMET: Communication-optimised multi-threaded error-detection technique. In *2016 International Conference on Compliers, Architectures, and Sythesis of Embedded Systems (CASES)*. IEEE, 1–10.

[46] Steven Muchnick et al. 1997. *Advanced compiler design implementation.* Morgan kaufmann.

[47] Shubhendu S. Mukherjee, Joel Emer, and Steven K. Reinhardt. 2005. The Soft Error Problem: An Architectural Perspective. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture (HPCA '05)*. 243–247.

[48] 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.

[49] Ben Nassi, Raz Ben-Netanel, Adi Shamir, and Yuval Elovici. 2019. Drones' Cryptanalysis-Smashing Cryptography with a Flicker. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1397–1414.

[50] Ben Nassi, Ron Bitton, Ryusuke Masuoka, Asaf Shabtai, and Yuval Elovici. 2021. SoK: Security and Privacy in the Age of Commercial Drones. In *Proc. IEEE Symp. Security Privacy (SP)*. 73–90.

[51] Muhammad Kashif Naveed and Hui Wu. 2020. Aster: Multi-bit Soft Error Recovery Using Idempotent Processing. *IEEE Transactions on Emerging Topics in Computing* 8, 4 (2020).

[52] Nahmsuk Oh, Philip P Shirvani, and Edward J McCluskey. 2002. Error detection by duplicated instructions in super-scalar processors. *IEEE Transactions on Reliability* 51, 1 (2002), 63–75.

[53] Steven K Reinhardt and Shubhendu S Mukherjee. 2000. *Transient fault detection via simultaneous multithreading*. Vol. 28. ACM.

[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] Eric Rotenberg. 1999. AR-SMT: A microarchitectural approach to fault tolerance in microprocessors. In *Digest of Papers. Twenty-Ninth Annual International Symposium on Fault-Tolerant Computing (Cat. No. 99CB36352)*. IEEE, 84–91.

[56] C. Sakalis, C. Leonardsson, S. Kaxiras, and A. 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)*. 101–111. https://doi.org/10.1109/ISPASS.2016.7482078

[57] Martin Schoeberl, Sahar Abbaspour, Benny Akesson, Neil Audsley, Raffaele Capasso, Jamie Garside, Kees Goossens, Sven Goossens, Scott Hansen, Reinhold Heckmann, et al. 2015. T-CREST: Time-predictable multi-core architecture for embedded systems. *Journal of Systems Architecture* 61, 9 (2015), 449–471.

[58] Martin Schoeberl, Pascal Schleuniger, Wolfgang Puffitsch, Florian Brandner, Christian W Probst, Sven Karlsson, and Tommy Thorn. 2011. Towards a time-predictable dual-issue microprocessor: The Patmos approach. In *Bringing Theory to Practice: Predictability and Performance in Embedded Systems*, Vol. 18. 11–21.

[59] Tingting Sha, Milo MK Martin, and Amir Roth. 2005. Scalable store-load forwarding via store queue index prediction. In *38th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'05)*. IEEE, 12–pp.

[60] Premkishore Shivakumar and Norman P Jouppi. 2001. Cacti 3.0: An integrated cache timing, power, and area model. (2001).

[61] Alex Shye, Tipp Moseley, Vijay Janapa Reddi, Joseph Blomstedt, and Daniel A Connors. 2007. Using process-level redundancy to exploit multiple cores for transient fault tolerance. In *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'07)*. IEEE, 297–306.

[62] Jared C Smolens, Brian T Gold, Babak Falsafi, and James C Hoe. 2006. Reunion: Complexity-effective multicore redundancy. In *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*. IEEE, 223–234.

[63] 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.

[64] 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.

[65] Daniel J Sorin, Milo MK Martin, Mark D Hill, and David A Wood. 2002. SafetyNet: improving the availability of shared memory multiprocessors with global checkpoint/recovery. In *Proceedings 29th Annual International Symposium on Computer Architecture*. IEEE, 123–134.

[66] Chikafumi Takahashi, Shinichi Shibahara, Kazuki Fukuoka, Jun Matsushima, Yuko Kitaji, Yasuhisa Shimazaki, Hirotaka Hara, and Takahiro Irita. 2016. 4.5 A 16nm FinFET heterogeneous nona-core SoC complying with ISO26262 ASIL-B: Achieving 10- 7 random hardware failures per hour reliability. In *2016 IEEE International Solid-State Circuits Conference (ISSCC)*. IEEE, 80–81.

[67] Gaurang Upasani, Xavier Vera, and Antonio Gonzalez. 2012. Setting an error detection infrastructure with low cost acoustic wave detectors.. In *ISCA*. 333–343.

[68] Gaurang Upasani, Xavier Vera, and Antonio Gonzalez. 2013. Reducing DUE-FIT of caches by exploiting acoustic wave detectors for error recovery.. In *IOLTS*. 85–91.

[69] Gaurang Upasani, Xavier Vera, and Antonio Gonzalez. 2014. Avoiding core's DUE & SDC via acoustic wave detectors and tailored error containment and recovery.. In *ISCA*. 37–48.

[70] Gaurang Upasani, Xavier Vera, and Antonio Gonzalez. 2015. A Case for Acoustic Wave Detectors for Soft-Errors. *IEEE Trans. Comput.* XX, 99 (2015).

[71] Gaurang Upasani, Xavier Vera, and Antonio Gonzalez. 2016. A case for acoustic wave detectors for soft-errors. *IEEE Trans. Comput.* 65, 1 (2016), 5–18.

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

[73] Vanchinathan Venkataramani, Bruno Bodin, Aditi Kulkarni, Tulika Mitra, and Li-Shiuan Peh. 2020. Time-Predictable Software-Defined Architecture with Sdf-Based Compiler Flow for 5g Baseband Processing. In *ICASSP 2020-2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 1553–1557.

[74] Cheng Wang, Ho-seop Kim, Youfeng Wu, and Victor Ying. 2007. Compiler-managed software-based redundant multi-threading for transient fault detection. In *International Symposium on Code Generation and Optimization (CGO'07)*. IEEE, 244–258.

[75] 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.

[76] Yun Zhang, Soumyadeep Ghosh, Jialu Huang, Jae W Lee, Scott A Mahlke, and David I August. 2012. Runtime asynchronous fault tolerance via speculation. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*. 145–154.

[77] Yun Zhang, Jae W Lee, Nick P Johnson, and David I August. 2012. DAFT: decoupled acyclic fault tolerance. *International Journal of Parallel Programming* 40, 1 (2012), 118–140.

[78] LinLin Zhu, Hui Guan, and ChuiJie Wu. 2015. A study of a three-dimensional self-propelled flying bird with flapping wings. *SCIENCE CHINA Physics, Mechanics & Astronomy* 58, 9 (2015), 1–16.